Władysław Homenda

Formal languages and automata

Lecture notes
Warszawa, 2010

Faculty of Mathematics and Information Science
Warsaw University of Technology

WARSAW UNIVERSITY OF TECHNOLOGY
DEVELOPMENT PROGRAMME

HUMAN CAPITAL
NATIONAL COHESION STRATEGY

EUROPEAN UNION
EUROPEAN
SOCIAL FUND

Contents

Acronyms

ε-NFA	finite automata with ε-moves (the class of)
ALL	all languages (the class of)
CFG	context-free grammars (the class of)
CFL	context-free languages (the class of)
CSG	context-sensitive grammars (the class of)
CSL	context-sensitive languages (the class of)
DFA	deterministic finite automata (the class of)
DPDA	deterministic pushdown automata (the class of)
DTM	deterministic Turing machines (the class of)
LBA	linear bounded automata (the class of)
NDFA	nondeterministic finite automata (the class of)
NPDA	nondeterministic pushdown automata (the class of)
NTM	nondeterministic Turing machines (the class of)
PDA	pushdown automata (the class of)
REL	recursively enumerable languages (the class of)
RgL	regular languages (the class of)
RkL	recursive languages (the class of)
TM	Turing machines (the class of)

Chapter 1
Regular expressions and regular grammars

Regular languages form the simplest class of formal languages. They are inductively defined by regular expressions. Regular languages also outline simple algebraic structures in the set of all words over a given alphabet. Due to this simplicity, they can be easily distinguished and identified. Regular languages are also generated by regular grammars and - what will be discussed in Chapter 6 - they are accepted by finite automata.

In this Chapter, fundamental properties of regular expressions and regular languages are studied. We cover a discussion on the following topics: operation on regular expressions, algebraic properties of the relation induced by regular languages (so called Myhill-Nerode lemma, which is a part a part of Myhill-Nerode theorem), structure of words of regular languages (pumping lemma). The study is supplemented by a section focused on regular grammars. All these topics are revisited in Chapter 7.

1.1 Regular expressions and regular languages

1.1.1 Regular expressions

The definition of regular expression is inductive, i.e. basic regular expressions are defined explicitly while more complex regular expressions could be constructed according to given rules.

Definition 1.1. Regular expression over an alphabet Σ is a construct defined as follows:

- Φ is a regular expression,
- ε is a regular expression,
- for each a in Σ, **a** is a regular expression,
- if r and s are regular expressions then

– $(r+s)$, the sum of regular expressions,
– (rs), the concatenation of regular expressions and
– $(r)^*$, the Kleene closure of the regular expression

are regular expressions.

Definition 1.2. Regular expressions over an alphabet Σ generate languages:

- Φ generates the empty language $\emptyset$:
- ε generates the language $\{\varepsilon\}$,
- for each a in Σ, **a** generates the language $\{a\}$,
- if regular expressions r and s generate languages R and S then
  - $(r+s)$ generates the language $R \cup S$ (union of languages R and S),
  - (rs) generates the language $(R \circ S)$ (concatenation of languages R and S),
  - $(r)^*$ generates the language R^* (Kleene closure of the language R).

Remark 1.1. Regular expressions are finite constructs. In fact, they are words over the alphabet $\Sigma \cup \{+, \circ, ^*, (,)\}$, where Σ is an alphabet of a regular expression. On the other hand, languages generated by regular expressions can be infinite.

Remark 1.2. Regular expressions and formulas describing languages generated by regular expressions include a large number of brackets, what makes them hardly readable. On the other hand, algebraic operators, logic operators as well as set theoretic operators are assumed to have priorities. This assumption allows dropping most of brackets of algebraic, logic as well as set theoretic expressions. By analogy, the same simplification is assumed for regular expressions. It is assumed that the sum operator has the lowest priority, concatenation has higher priority and the Kleene operator has the highest priority. The same assumption is adapted for operators on languages. Union has the lowest priority, concatenation has higher priority and the Kleene closure is of the highest priority. We will drop any pair of brackets if it does not change the order of operators when priorities are applied.

Remark 1.3. Regular expressions can be interpreted as strings of symbols. Therefore, a simplified form of a regular expression is not equal to its original form in terms of strings' equality. Yet, languages generated by both forms of a regular expression are identical. So then, both forms are considered to be equivalent. From now on, if not stated otherwise, equivalent regular expressions will be considered to be equal.

1.1.2 Regular languages

Definition 1.3. Regular languages are those and only those generated by regular expressions.

Remark 1.4. The following equalities hold (in terms of Remark 1.3).

1. $\emptyset + r = r + \emptyset = r$
2. $\emptyset r = r\emptyset = \emptyset$
3. $\varepsilon r = r\varepsilon = r$
4. $\varepsilon + r = r + \varepsilon$
5. $r + s = s + r$
6. $(r + s) + t = r + (s + t) = r + s + t$
7. $(rs)t = r(st) = rst$
8. $r(s + t) = rs + rt$
9. $(r + s)t = rt + st$
10. $(r^*)^* = r^*$
11. $(r^*s^*)^* = (r + s)^*$
12. $(r^* + s^*)^* = (r + s)^*$

1.1.3 The Myhill-Nerode lemma

The Myhill-Nerode theorem is the most important tool characterizing regular languages. The theorem is formulated and proved in Chapter 7. In this Chapter a limited version of the Myhill-Nerode theorem is formulated, which will be referred to as the Myhill-Nerode lemma. The proof of the lemma is a direct consequence of a proof of the Myhill-Nerode theorem. The lemma is not proved here, since important topics used in the proof have not been discussed yet. The lemma is formulated here since it is a crucial tool used in the identification of regular languages. It will be used in this and other Chapters.

Lemma 1.1. ***(the Myhill-Nerode lemma)*** *A language L is regular if and only if the relation R_L induced by the language L has a finite number of equivalence classes.*

1.1.4 The pumping lemma

The pumping lemma is another tool - in addition to regular expressions and the Myhill-Nerode lemma - that can be used to characterize regular languages. Regular expressions and the Myhill-Nerode lemma (and regular grammars, which are discussed in the next section) are aimed at proving that a language is regular. The Myhill-Nerode lemma and the pumping lemma can be used in proving that a language is not regular. As in case of the Myhill-Nerode lemma, the pumping lemma is not proved here, since important topics used in the proof have not been discussed yet. The lemma is formulated here since it is a crucial tool used in the identification of regular languages. It will be proved in Chapter 7.

Lemma 1.2. ***(the pumping lemma for regular languages)***
If a language L is regular then there exists a constant n_L such that for any word the following condition holds:

$$(|z| \geq n_L) \Rightarrow [(\bigvee_{u,v,w} z = uvw \wedge |uv| \leq n_L \wedge |v| \geq 1) \bigwedge_{i=0,1,2,\ldots} z_i = uv^i w \in L]$$

The pumping lemma formulates conditions necessary for a regular language. It shows that nature of regular languages is finite and length of words of a given regular language is limited by some constant n_L determined by the pumping lemma. If a regular language includes a word of length greater that or equal to n_L, then it is an infinite language. However, a structure of words that are longer than or equal to n_L is fairly simple. Such words are generated by inserting strings of length limited by the constant n_L into words of the language that are shorter than n_L. In other words, any word of a regular language, not shorter than n_L, have a floating part to be deleted leaving the remaining part in the language.

Since the pumping lemma formulates necessary conditions, in its direct form it is of limited importance. It can be used for analysis of a structure of words of the language. If words satisfy conclusion of the pumping lemma, then the language could be intuitively presumed to be regular. Then, based on such a supposition, the language could be proved to be regular. In practice, we use contraposition of the pumping lemma rather than its generic version. Contraposition of the pumping lemma formulates sufficient conditions for a language not to be regular. This makes contraposition of the pumping lemma to be extremely useful in proving that certain languages are not regular.

Lemma 1.3. *If for any constant n_L there exists a word $z \in L$ such that*

$$(|z| \geq n_L) \wedge [(\bigwedge_{u,v,w} z = uvw \wedge |uv| \leq n_L \wedge |v| \geq 1) \bigvee_{i=0,1,2,\ldots} z_i = uv^i w \notin L]$$

then a language L is not regular.

1.1.5 Regular grammars

Regular grammars constitute the simplest class of grammars. They generate regular languages, so then they can be used for proving that languages are regular. A proof that regular grammars generate regular languages will be provided in Chapter 7. In this section, we provide a definition of regular grammars and use it as a tool for processing regular languages.

Definition 1.4. A grammar $G = (V, T, P, S)$ is:

- left-linear if and only if all its productions of the form $A \to Bw$ or $A \to w$, where A, B are nonterminals, i.e. $A, B \in V$, w is a string of terminals, possibly empty, i.e. $w \in T^*$,
- right-linear if and only if all its productions of the form $a \to wB$ or $A \to w$, where $A\ B$ are nonterminals, w is a string of terminals, possibly empty,item regular if and only if it is either left-linear, or right linear.

Remark 1.5. Every left-linear grammar has equivalent right-linear grammar and every right linear grammar has equivalent left-linear grammar. Equivalence of grammars is understood as identity of generated languages. This observation will be proved in Chapter 7.

Chapter 2
Context-free grammars

The class of context-free languages is a next simplest class of languages, beside the class of regular languages. Context-free languages are generated by context-free grammars, which correspond to regular grammars and regular expressions. Context-free grammars are one of a few most important tools used for analysis of context-free languages. They are incomparably simpler than context-sensitive and unrestricted grammars. On the other hand, they define a wider class of languages than simpler regular grammars. The structure of words they generate is rich. Moreover, context-free grammars are well elaborated. They provide effective methods of analysis and generation of languages. Furthermore, there are effective methods of automatic analysis and processing of context-free grammars. Due to these advantages they are widely applied in practice, e.g. in natural language processing, processing of programming languages, translation of formal languages, pattern recognition, etc.

Algebraic structures of context-free languages created in the set of all words over an alphabet are much more complex than those created by regular languages. For this reason algebraic analysis of context-free languages is limited. For instance, there is no tool corresponding to the Myhill-Nerode theorem.

This Chapter is devoted to a discussion of basic properties of context-free languages, especially those properties, which arise out of the analysis of context-free grammars. Let us emphasize that an analysis of context-free grammars is well established given their practical relevance.

2.1 Context-free grammars - basics

Context-free grammars have a simple form of productions: a nonterminal creates production's left hand side while a sequence of terminals and nonterminals figures its right hand side. This form of productions allows for a simple illustration of derivation. A derivation of a word can be demonstrated as a tree, what easies the proofs of such important properties as the pumping lemma, the Ogden lemma and a decision algorithm for context-free-languages.

Definition 2.1. A grammar $G = (V, T, P, S)$ is a context-free grammar if and only if left side of any its production is a nonterminal, i.e. any $p \in P$ is of a form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$.

Definition 2.2. A language $L \subset \Sigma^*$ is context-free if and only if it is generated by a context-free grammar.

For context-free grammar, a derivation of a word can also be presented in the form of a tree.

Definition 2.3. A derivation tree in a context-free grammar $G = (V, T, P, S)$ is a tree $T = (w, e \subset W \times W)$ satisfying the following properties:

1. it is compatible with inductive definition of tree, but the set of children of every vertex is ordered,
2. every vertex $w \in W$ of the tree is labelled with a nonterminal symbol or terminal symbol or empty word: $v \in V \cup T \cap \{\varepsilon\}$,
3. the root of the tree is labelled with the initial symbols S of the grammar,
4. internal vertices of the tree (i.e. vertices different than leaves) are labelled with nonterminals,
5. if a nonterminal A labels an internal vertex and children of this vertex are labelled with symbols (terminals, nonterminals or the empty word) $X_1 X_2 \ldots X_k$ in this order, then there exists a production $A \to X_1 X_2 \ldots X_k$,
6. a vertex labelled with the empty word is the only child of its parent.

We say that a word is ambiguous if it has more than a single derivation tree. Equivalently, we can say that a word is ambiguous if and only if it has more than one leftmost derivation, or if it has more than one rightmost derivation. A context-free grammar is said to be ambiguous if and only if there is an ambiguous word. A context-free language is inherently ambiguous if and only if its every context-free grammar is ambiguous.

2.2 Simplification of context-free grammars

The definition of context-free grammars does not include optimization mechanisms. In this section we discuss some methods of simplification of context free grammars. For instance, a context-free grammar including symbols and productions, which are never used in derivation of any word, may be transformed to a form simpler to use. Also, a context-free grammar can be restructured to a form more suitable for a given application

2.2.1 Useless symbols

In particular, a context-free grammar can include symbols that are *useless* for generated language . A set of nonterminals which will never produce a sequence of terminals is the first type of useless symbols. A set of symbols (terminals or nonterminals) never used in derivation from initial symbol of the grammar forms the second type of useless symbols. Both types of useless symbols can be removed together with productions including such symbols (such productions are invalid in terms of reduced sets of nonterminals and terminals). Both grammars, the initial one and the grammar with removed useless symbols and removed invalid productions, generate the same language. This observation is obvious since a production, which include useless symbols can never be used in a derivation of any word over the terminal alphabet.

Proposition 2.1. *For any context-free grammar $G = (V, T, P, S)$ generating a nonempty language $L(G)$ there exists an equivalent (i.e., generating the same language) context-free grammar $G' = (V', T, P', S)$ such that every nonterminal $A \in V'$ generates a sequence of terminals (possibly empty). Note that an initial symbol is useless if a context-free grammar generates the empty language. The following algorithm shows how to remove useless symbols of the first type:*

```
begin
 V_old := V_new := ∅
 for each production p: A → α, p∈P do
 if α ∈ T* then V_new := V_new ∪ {A}
      {start with nonterminals producing a string of terminals}
 while V_old ≠ V_new do
 begin
 V:_old := V_new
 for each production p: A → α, p∈P do
 if α ∈ (T ∪ V_old)* then V_new := V_new ∪ {A}
 end
 V' := V_new   {new set of nonterminals}
 P' := {A → α ∈ P : A ∈ V', supp(α) ⊂ (V' ∪ T)}}
   {nonterminals of productions must be included into new set of nonterminals}
end
```

Proposition 2.2. *For any context-free grammar $G = (V, T, P, S)$ generating a nonempty language $L(G)$ there exists an equivalent (i.e., generating the same language) context-free grammar $G'' = (V'', T'', P'', S)$ such that every nonterminal symbol $A \in V''$ and every terminal symbol $a \in T''$ could be derived from the initial symbol of the grammar. The following algorithm allows for removing useless symbols of the second type:*

```
begin
 V_old := V_new := ∅
```

```
V_new:={S}, T_new:=∅
while V_old ≠V_new or T_old ≠T_new" do"
begin
V:_old:=V_new, T_old:=Tnew
for A∈V_old do
begin
 V_new:=V_new∪{B∈V : exists A→α and B∈supp(α)}
 T_new:=T_new∪{a∈T : exists A→α and a∈supp(α)}
end
end
V'':=V_new   {new set of nonterminals}
T'':=T_new   {new set of terminals}
P'':={A→α ∈ P : A ∈ V', supp(α)⊂(V''∪T'')}}
   {nonterminals of productions must be included into new set of nonterminals}
end
```

2.2.2 Nullable symbols and ε-productions

Elimination of ε-productions and nullable symbols is the next step of simplification of context-free grammars. ε-production is a production of the form $A \to \varepsilon$ and nullable symbol is a nonterminal symbol producing the empty word $A \to^* \varepsilon$. Of course, if the empty word is derivable in a context-free grammar G, then it is not possible to eliminate all ε-production and nullable symbols. However, removing ε-productions and nullable symbols from the grammar G turns it to the grammar generating the language $L(G) - \{\varepsilon\}$. Therefore the process of elimination of nullable symbols and ε-production will be applied to context-free grammars not producing the empty word.

The following algorithm finds nullable symbols in a context-free grammar $G = (V, T, P, S)$:

```
begin
 V_old:=∅   {begin with no nullable symbols}
 V_new:={A∈V : A→ε is a production}
       {add all nonterminals producing directly the empty word}
 while V_old ≠V_new do
 begin
 V:_old:=V_new
 V_new:=V_new∪{A∈V : exists A→α, where α∈V*_old}
    {add all nonterminals producing directly a word over nullable symbols}
 end
end
```

Having the set of nullable symbols we will be able to remove ε-productions. Observe that nullable symbol in a production (its right hand side) either can generate a string of terminal symbols, or can be turned to the empty word. As a consequence, a nullable symbol can either be left in the right side of a production (when it produces a nonempty sequence of terminal symbols), or it can be dropped from the right hand side of a production (when it generates the empty word). This observation leads to the following method.

Proposition 2.3. *Let $G = (V, T, P, S)$ is a context-free grammar with no useless symbols generating a language without the empty word. If $A \to X_1 X_2 \ldots X_n$ is a production, then this production is replaced with a set of all productions of a form $A \to \alpha_1 \alpha_2 \ldots \alpha_n$ satisfying conditions:*

- *$\alpha_i = X_i$, if X_i is not nullable (i.e., it is a terminal symbol or a not nullable nonterminal symbol), for all $i = 1, 2, \ldots, n$,*
- *$\alpha_i = X_i$ or $\alpha_i = \varepsilon$, if X_i is nullable, for all $i = 1, 2, \ldots, n$, (i.e. we get two productions, one with X_i at the right hand side and another one without X_i),*
- *not all $\alpha_1, \alpha_2, \ldots, \alpha_n$ are equal to the empty word (this condition eliminates ε-productions). Of course, the existing ε-productions are removed.*

Notice that this method turns status of nullable nonterminals to not nullable ones rather then removing them from the grammar. Finally, we obtain a grammar $G' = (V, T, P', S)$ without nullable symbols and ε-productions. This grammar has a modified set of productions and is equivalent to the grammar G, i.e., generates the same language.

Proof. Proof of equivalence of both grammars is based on equivalence of derivations in both grammars G and G':

- both grammars have the same sets of terminals and nonterminals,
- a derivation in the grammar G of any word $w \in L(G)$ can be turned to a derivation of the same word in the grammar G':
  - if a derivation does not employs ε-productions, then this is also a derivation in the grammar G',
  - if an ε-production $X \to \varepsilon$ is applied in a part of derivation with a production $Y \to \alpha X \beta$ utilized prior to this $X \to \varepsilon$ ε-production:

$$\ldots \to \gamma Y \delta \to \gamma \alpha X \beta \delta \to \gamma \alpha \beta \delta \to \ldots$$

    then this part can not be included in any derivation in G', but it can be turned to a fragment of a derivation in G' shown below. Here the production $Y \to \alpha\beta$ with the nullable symbol X dropped is employed,

$$\ldots \to \gamma Y \delta \to \gamma \alpha \beta \delta \to \ldots$$

    Finally, if we apply analogous replacement for every ε-production, the derivation of the word $w \in L(G)$ in the grammar G is turned into a derivation of the same word in the grammar G',

- a derivation in the grammar G' of any word $w \in L(G')$ can be turned to a derivation of the same word in the grammar G:
  - if a derivation utilizes only productions of the grammar G, then it is a valid derivation of the word w in G,
  - otherwise, a derivation employs at least one production of a form $Y \to \alpha\beta$ of the grammar G' gotten from a production of a form $Y \to \alpha' X \beta'$. Then a part of derivation
  $$\ldots \to \gamma Y \delta \to \gamma\alpha\beta\delta \to \ldots$$
  can be replaced by a fragment of a derivation in a grammar G with inserted a series of ε-productions:
  $$\ldots \to \gamma Y \delta \to \gamma\alpha' X \beta' \delta \to \ldots \to \gamma\alpha\beta\delta \to \ldots$$
  where $X \to \varepsilon$ is an ε-productions in the grammar G and string α and β are gotten by applying the same scheme to all nullable symbols of both strings. This method applied to all productions of the derivation of the word $w \in L(G')$ in the grammar G' turns this derivation to a derivation of the same word in the grammar G.

Finally, comparing languages generated by a context-free grammar G and by its transformed form G' without nullable symbols and ε-productions, we can state that $L(G') = L(G) - \{\varepsilon\}$.

2.2.3 *Unit productions*

A context-free grammar $G = (V, T, P, S)$ may have *unit productions*. Unit productions are of a form $A \to B$, where $A, B \in V$. Unit productions are confusing and do not provide any new abilities for language generation. Elimination of unit productions is the next step of grammar simplification. The method of removing unit productions is concerned with substituting a unit production $A \to B$ with a series of productions $A \to \alpha_i$ for all B-productions $B \to \alpha_i$. The method is outlined in the form of the following algorithm:

```
begin
 while there exists a unit production A→ B do
 begin
 if (A=B) then remove the production
 else
 replace the production A→ B with
 productions A→ γ1 | ... | γk
 where γ1, ..., γk are right hand side
 of all B-productions
 end
```

```
end
```

The new grammar $G''' = (V, T, P''', S)$ produced by this algorithm may have useless nonterminal symbols, which need to be removed. For instance, the grammar

$$G = (\{S, A, B\}, \{a\}, \{S \to A|a, A \to B, B \to a\}, S)$$

is turned to the grammar without unit production, but with useless nonterminal symbols A and B:

$$G''' = (\{S, A, B\}, \{a\}, \{S \to a, A \to a, B \to a\}, S)$$

Note that elimination of the unit production $S \to A$ introduces the production $S \to a$, which already exists in the grammar. The process of removing useless symbols yields the following grammar:

$$G^* = (\{S\}, \{a\}, \{S \to a\}, S)$$

A grammar without unit productions is equivalent to the former one. To prove this, let us consider a derivation tree in the former grammar. If a part of derivation tree matching a derivation $A_{i_1} \to A_{i_2} \to \cdots \to A_{i_r} \to \alpha$ (the last production is not unit, the vertex A_{i_r} has at least two children or one leaf) includes a repeating nonterminal in the path $A_{i_1} \to \cdots A_{i_p} \to A' \to \cdots \to A' \to A_{i_r} \to \cdots A_{i_r}$, then this path can be shortened to $A_{i_1} \to \cdots A_{i_p} \to A' \to A_{i_r} \to \alpha$.

Let assume that a derivation $A_{i_1} \to A_{i_2} \to \cdots \to A_{i_r} \to \alpha$ does not have a repeating nonterminal. The method of elimination of unit productions provides a production $A_{i_1} \to \alpha$, so then this derivation can be cut to $A_{i_1} \to \alpha$ which creates a part of a derivation tree in the grammar G'''. And vice versa, if we have a part of derivation tree in the grammar G''' matching a production $A_{i_1} \to \alpha$ in the grammar G''', then this production either belongs to the grammar G or it can be turned to a derivation $A_{i_1} \to A_{i_2} \to \cdots \to A_{i_r} \to \alpha$ in the grammar G'''.

In conclusion, a derivation tree in the grammar G can be turned to a derivation tree in the grammar G''' by replacing all such transformations. And vice versa, a derivation tree in the grammar G''' can be turned to a derivation tree in the grammar G.

2.3 Normal forms of context-free grammars

We discuss two normal forms of context-free grammars, that is Chomsky normal form and Greibach normal form. Conversions of context-free grammars to normal forms come as a further step in the simplification of grammars. Normal forms are grammars with restrictions put on the form of productions. Grammars in normal forms produce languages without the empty word, so then only grammars not generating the empty word could be transformed to normal forms. This is why a context-free grammar, in which the empty word is derivable, should be turned to a form

generating the same language without the empty word. Since both normal forms do not admit ε-productions, a removal of ε-productions and nullable symbols will convert any context-free grammar to a form generating the language without the empty word. Normal forms do not necessarily require removal of useless symbols. Anyway, it is recommended to simplify a grammar by removing useless symbols first.

2.3.1 Chomsky normal form

Definition 2.4. A context-free grammar $G = (V, T, P, S)$ is in Chomsky normal form if and only if its productions are of the form $A \to BC$ or $A \to a$, where $A, B, C \in V$, $a \in T$ (i.e. any production turns a nonterminal to two nonterminals or to one terminal).

Proposition 2.4. *Any context-free grammar without ε-productions and unit productions can be transformed to Chomsky normal form.*

Proof. let assume that a context-free grammar $G = (V, T, P, S)$ does not have ε-productions and unit productions, so then a right-hand side of any production is a terminal symbol or is a string of at least two symbols. Productions with one terminal symbol on right hand side are in Chomsky normal form. Such productions will not be changed.

Productions having at least two symbols on the right hand side will be transformed to a set of productions according to rules:

1. every production of a form $A \to \alpha_1 a \alpha_2$ is substituted with two productions $A \to \alpha_1 A' \alpha_2$ and $A \to a$, where: $A \in V$, $a \in T$, $\alpha_1 \alpha_2 \in (V \cup T)^+$ and A' is a new nonterminal,
2. every production of a form $A \to A_1 A_2 \ldots A_n$, $n > 2$ is substituted with two productions $A \to A_{1,2} \ldots A_n$ and $A_{1,2} \to A_1 A_2$, where $A, A_1, A_2, \ldots, A_n \in V$, $A_{1,2}$ is a new nonterminal.

The new grammar $G' = (V', T, P', S)$ includes newly added nonterminals. All productions of the grammar G not in Chomsky form are replaced with sets of new productions in Chomsky form.

Both grammars G and G' are equivalent, i.e., they generate the same language. To show this, let us consider a derivation tree of a word in grammars G and G':

- a local fragment of a derivation tree in the grammar G matching a production of a form $A \to \alpha_1 a \alpha_2$ is shown in part (i) of Figure 2.1. This fragment can be replaced with a fragment matching two productions $A \to \alpha_1 A' \alpha_2$ and $A \to a$ (both substitute the former production) as shown in part (ii) of Figure 2.1. The new tree generates the same crop (although it may not be a valid tree in any of these two grammars). On the other hand, a production $A \to \alpha_1 A' \alpha_2$ of the grammar G' forces the production $A' \to a$ since the nonterminal symbol A' is

unique in the grammar G' and appears only in former two productions. Both former productions correspond to a part of a derivation tree in the grammar G' shown in part (ii) of Figure 2.1. This part can be turned to a structure shown in part (i) of Figure 2.1, which corresponds to a production $A \to \alpha_1 a \alpha_2$ of the grammar G,

- parts of derivation trees corresponding to a production of a form $A \to A_1 A_2 \ldots A_n$ and to its substitutions $A \to A_{1,2} \ldots A_n$ and $A_{1,2} \to A_1 A_2$ is shown in parts (iii) and (iv) of Figure 2.1,
- finally,
  - substituting all fragments of a derivation tree shown in parts (i) and (iii) of Figure 2.1 turns a derivation tree in the grammar G to a derivation tree in the grammar G',
  - opposite substitutions turns a derivation tree in the grammar G' to a derivation tree in the grammar G,
  - substitutions does not change the crop of subjected trees,

what justifies equivalence of grammars G and G'.

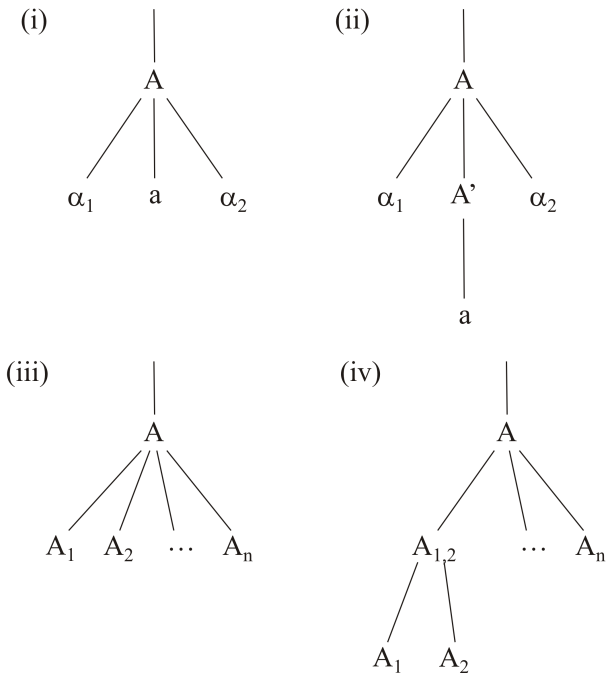

Fig. 2.1 Equivalence of a context-free grammar and its Chomsky normal form.

2.3.2 Greibach normal form

Definition 2.5. A context-free grammar $G = (V, T, P, S)$ is in Greibach normal form if and only if its productions are of the form $A \to a\alpha$ where $A \in V$, $a \in T$, $\alpha \in V^*$ (i.e. any production turns a nonterminal to a terminal and a string (possibly empty) of nonterminals.

Proposition 2.5. *Any context-free grammar without ε-productions and unit productions could be transformed to Greibach normal form.*

Proof. The following method can be used for transformation of a context-free grammar $G = (V, T, P, S)$ to Greibach normal form:

1. transform the grammar to Chomsky normal form $G_C = (V_C, T, P_C, S)$,
2. enumerate nonterminal symbols in $V_C = \{A_1, A_2, \ldots, A_n\}$,
3. modify the grammar G_C such that the right hand side of every A_i-production begins with a terminal symbol or with a nonterminal symbol of a higher index:

$$(*) \quad \begin{cases} A_i \to a\alpha \\ A_i \to a\beta \end{cases} \quad \text{or} \quad \text{where } a \in T, \ \alpha, \beta \in V_C$$

Let assume that for $i = 1, 2, 3, \ldots, k-1$ all A_i productions satisfy the condition $(^*)$ given above and that some A_k production do not satisfy this condition. If a production $A_k \to A_j\,\alpha$, where $A_k, A_j \in V_C$ and $\alpha \in V_C^*$, does not satisfy $(^*)$, then either (i) $k > j$ or (ii) $k = j$,

(i) for every A_j-production replace the production $A_k \to A_j\,\alpha$ with the production $A_k \to \beta_l\alpha$, where β_l is the right-hand side of the A_j-production. The right-hand sides of new productions $A_k \to \beta_l\alpha$ either begins with a terminal or with a nonterminal A_l with index greater than j. Observe that every such substitution increases the value of the index of the right hand side nonterminal. Repeating substitutions at most $k-1$ times, we obtain all A_k production with right-hand side begin with either a terminal symbol A_k or with the nonterminal symbol or with a nonterminal symbol A_l with $l > k$, so then the case (i) is eliminated,

((ii) let that there is the following set of A_k-productions:

$$A_k \to A_k\,\alpha_1 \mid A_k\,\alpha_2 \mid \ldots \mid A_k\,\alpha_p \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_r$$

where: $\alpha_1, \ldots, \alpha_p \in V_C^*$, $\beta_1, \ldots, \beta_r \in (T \cup \{A_{k+1}, \ldots A_n\}) \circ V_C^*$
i.e. β_l is a terminal or a nonterminal with index greater than k followed by a sequence (possibly empty) of nonterminals, i.e. all A_k-productions satisfy the condition (ii),
This set of A_k-productions is replaced with the following set of new productions:

$$A_k \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_r \mid \beta_1 B_k \mid \beta_2 B_k \mid \ldots \mid \beta_r B_k$$
$$B_k \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_p \mid \alpha_1 B_k \mid \alpha_2 B_k \mid \ldots \mid \alpha_p B_k$$

where: B_k, is a new nonterminal. Nonterminals B_i are ordered according to their indexes and are followed by all nonterminals A_i.
Notice that all newly included productions satisfy the condition $(^*)$. Therefore, the condition $(^*)$ is satisfied for all A_i-productions and B_i-productions for $i = 1, 2, \ldots, k$,

4. the process of the recent point repeated for successive nonterminals guarantee satisfaction of the condition $(^*)$ for all productions. Moreover, the right hand side of every A_n-production must begin with a terminal symbol since A_n is the last nonterminal in the introduced order, i.e., every A_n-production is in the Greibach normal form,
5. for backward values of $k = n-1, n-2, \ldots, 2, 1$, for every A_k-production with right-hand side with a leading nonterminal symbol A_j, $j \in \{n, n-1, \ldots, k+1, k+1\}$, replace this production with a new set of productions. Productions of this new set are obtained by replacing the leading nonterminal A_j with right hand sides of A_j-productions. We do the same with B_j-productions for decreasing values of index j. Productions of the newly created set are in Greibach normal form because all A_j-productions and B_j-productions have already been turned to Greibach normal form,
6. as a result, we get a grammar $G_G = (V_G, T, P_G, S)$ in Greibach normal form, where V_G is a set of nonterminal symbols V_C supplemented with nonterminal symbols B_l created in point 3 (ii).

It has already been noted that the conversion to Chomsky normal form does not change the language being generated. The transformation described in point 3 (i) also keeps the generated language without any changes - justification is the same as for conversion to Chomsky normal form and for elimination of unit productions.

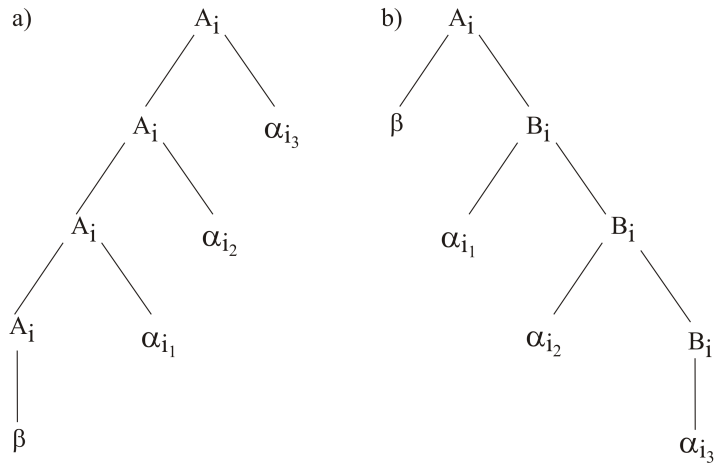

Fig. 2.2 Invariability of unlooping method of a context-free grammar.

Now we justify that elimination of looped productions in point 3 (ii) does not change the language. Let assume that a part of a derivation tree in the grammar G employs several productions of a form $A_i \to A_i\alpha_j$, i.e. a nonterminal symbol A_i is substituted by a string of nonterminal symbols $A_i\alpha_i$ several times and finally A_i is substituted by a string β. The following example concerning a fragment of a derivation tree in the grammar G shown in part a) of Figure 2.2 is considered. This fragment is equivalent to the following derivation G:

$$A_i \to A_i\alpha_{i_3} \to A_i\alpha_{i_2}\alpha_{i_3} \to A_i\alpha_{i_1}\alpha_{i_2}\alpha_{i_3} \to \beta\alpha_{i_1}\alpha_{i_2}\alpha_{i_3}$$

The rules in point 3 (ii) of Proposition 2.5 employed to the above fragment of derivation tree produces a fragment of a derivation tree in the grammar G_G. This fragment is shown in Figure 2.2 (b). It is equivalent to the following derivation completed in the grammar G_G:

$$A_i \to \beta B_i \to \beta\alpha_{i_1}B_i \to \beta\alpha_{i_1}\alpha_{i_2}B_i \to \beta\alpha_{i_1}\alpha_{i_2}\alpha_{i_3}$$

Notice that derivations presented here may be distorted by other productions. Nevertheless, the altered derivations are equivalent to the same derivation trees.

2.4 Pumping and Ogden lemmas

The pumping lemma for context-free languages and the Ogden lemma characterize the structure of words of context-free languages. Both lemmas are very important tools used in identification of context-free languages, like other tools in case of regular languages.

2.4.1 The pumping lemma

The pumping lemma formulates conditions necessary for a language to be context-free. It shows that the nature of context-free languages is finite and length of words of a given context-free language is limited by some constant n_L, whose value is determined in the pumping lemma. If a context-free language includes a word of length greater than or equal to n_L, then it is an infinite language. However, a structure of words that are longer than or equal to n_L, is fairly simple. Such words are generated by inserting strings of length limited by the constant n_L into words of the language that are shorter than n_L. We can also say that any word of a context-free language, not shorter than n_L, have two floating parts that can be deleted simultaneously leaving the remaining part in the language (let us recall that words of regular languages have only one part that can be subjected to deletion).

Lemma 2.1. ***the pumping lemma for context-free languages***
If a language is context-free
then there exists a constant n_L such that for any word $z \in L$ the following condition holds:

$$(|z| \geq n_L) \Rightarrow \left[\left(\bigvee_{u,v,w} z = uvwxy \wedge |vwx| \leq n_L \wedge |vx| \geq 1 \right) \bigwedge_{i=0,1,2,\ldots} z_i = uv^i wx^i y \in L \right]$$

Proof. If a language L is finite, then a constant n_L greater than the length of a longest word of this language, satisfies the lemma.

Consider the case of infinite languages. Let us assume that a context-free grammar $G = (V, T, P, S)$ in Chomsky normal form generates a (context-free) language L. Derivation trees in such a grammar are binary trees. We use the property that height (length of a longest path from the root to a leaf) in any binary tree with k leaves is not less than $\log_2 k$. All vertexes of such a path, except the last one, are labelled by nonterminal symbols of the grammar. The last vertex of this path is a leaf of the tree and is labelled by a terminal symbol.

Let $|V| = N$. If we set the constant $n_L = 2^{N+1}$, then derivation tree of a word z not shorter than n_L has height not less than $N+1$. Therefore there exists a path from the root S to a leaf not shorter than $N+1$. Consequently, $N+1$ vertexes of this path are labelled by N nonterminal symbols and the leaf is labelled by a terminal symbol a, c.f. Figure 2.3. As a result, there exist two (maybe more) vertexes labelled by the same nonterminal symbol. Let us consider the pair of vertexes symbol closest to the leaf a that are labelled by the same nonterminal symbol A. To distinguish labels of vertexes of this pair, the nonterminal symbol A is denoted A' and A'' respectively. The crop z of the derivation tree is divided into five parts: u, v, w, x and y. w is the crop of the subtree with the root A'' while vwx is the crop of the subtree with the root A'.

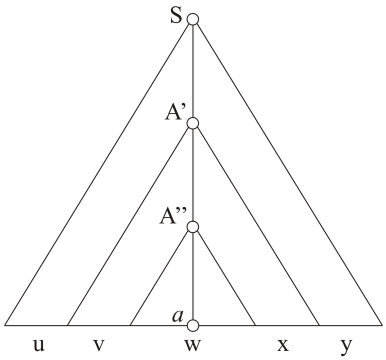

Fig. 2.3 The derivation tree of a word z of length $2^{|V|+1}$.

If we replace the subtree with the root A' by the subtree with the root A'', we will get a valid derivation tree with the crop $z_0 = uwy = uv^0wx^0y$. This tree is shown at

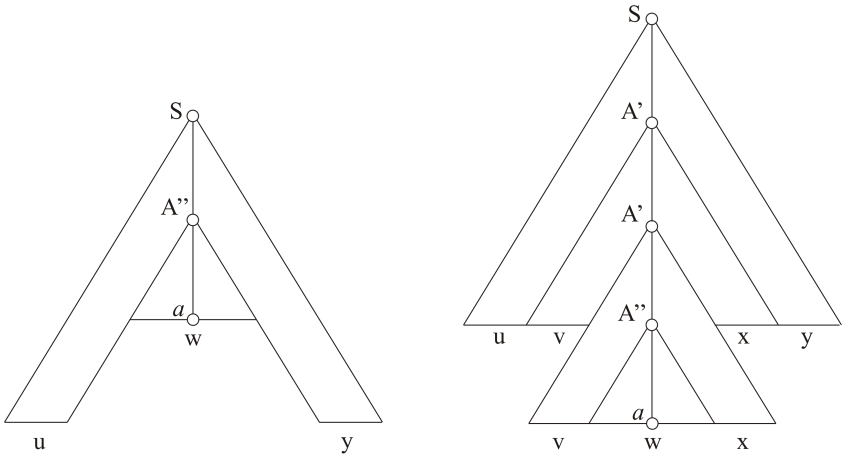

Fig. 2.4 The derivation trees obtained from the tree shown in Figure 2.3.

the left part of Figure 2.4. If we replace the subtree with the root A'' by the subtree with the root A', we will obtain the tree shown at the right part of Figure 2.4, which is still a valid derivation tree with the crop $z_2 = uvvwxxy = uv^2wx^2y$. Replacing the subtree with the root A'' in this last tree by the subtree with the root A', we get a derivation tree with the crop $z_3 = uvvvwxxxy = uv^3wx^3y$. This iterative process can be continued.

In this way, we show that the pumping lemma for context-free languages is satisfied. To fulfill formal requirements, mathematical induction should be applied based on the number of replacements of the subtree with the root A'' by the subtree with the root A'. The reader can elaborate details of an inductive proof.

Since the pumping lemma formulates necessary conditions, it is of limited importance in its direct form. It can be employed to analyze a structure of words of the language. If words satisfy the conclusion of the pumping lemma, then the language could be intuitively presumed to be context-free. Then, based on such a supposition, the language could be proved to be context-free. In practice, we use a contraposition of the pumping lemma rather than its generic version, like in case of the pumping lemma for regular languages. The contraposition of the pumping lemma formulates sufficient conditions for a language not to be context-free. This makes the contraposition of the pumping lemma to be extremely useful in proving that certain languages are not context-free.

Remark 2.1. Let us note that the pumping lemma for regular languages is a special case of the pumping lemma for context-free languages. The assumption that $uv = \varepsilon$ turns the pumping lemma for context-free languages to the pumping lemma for regular languages.

Lemma 2.2. *If for any constant n_L there exists a word $z \in L$ such that*

$$(|z| \geq n_L) \wedge \Big[\Big(\bigwedge_{u,v,w} z = uvwxy \wedge |vwx| \leq n_L \wedge |vx| \geq 1 \Big) \bigvee_{i=0,1,2,\ldots} z_i = uv^i wx^i y \notin L \Big]$$

then a language L is not context-free.

2.4.2 The Ogden lemma

The pumping lemma is a powerful tool used for proving that languages are not context-free. However, contraposition of the pumping lemma can be hardly applied for some types of languages. In such difficult cases the Ogden lemma may help in proving that a language is not context-free. In fact, the pumping lemma is a special case of the Ogden lemma. However, the pumping lemma is easier to be applied than the Ogden lemma. This is why the pumping lemma is used for simpler problems.

Lemma 2.3. ***the Ogden lemma***
If a language L is context-free,
then there exists a constant n_L such that for any word $z \in L$ and for at least n_L symbols marked in z there exists a split $z = uvwxy$ holding conditions:

- *vx includes at least one marked symbol,*
- *vwx includes no more than n_L marked symbols*

and such that $z_i = uv^i wx^i y \in L$ for any $i = 0, 1, 2 \ldots$.

Proof. Let us notice that height of a derivation tree is - of course - not less than $\log_2 n_L$. This means that there exists a path from the root to a leaf not shorter than $\log_2 n_L$. Such a path starts in the root and then, for every its vertex, goes to this child, which has no less marked leaves in its subtree, than the other child has. Having such a path we can do the same replacements in the derivation tree as we did in the proof of the pumping lemma.

Note that marking all symbols of a word we turn the Ogden lemma to the pumping lemma. Thus, the pumping lemma is a special case of the Ogden lemma.

As in case of the pumping lemma, the Ogden lemma formulates necessary conditions for a language to be context-free. This is why the Ogden lemma in its direct form is hardly applicable. In practice we use the contraposition of the Ogden lemma.

Lemma 2.4 (contraposition of the Ogden lemma).

if for any constant n_L there exists a word $z \in L$ with at least n_L symbols marked such that for any split $z = uvwxy$ holding conditions:

- *vx includes at least one marked symbol,*
- *vwx includes no more than n_L marked symbols*

and such that there exists a constant $i \in \{0, 1, 2, \ldots\}$ for which $z_i = uv^i wx^i y \notin L$ then the language is not context-free.

2.5 Membership of context-free languages

The central question is how to check if a word belongs to a language. Having a context-free grammar we can answer the question if a word is generated in this grammar. Moreover, we will be able to construct a derivation of a given word if it is generated in the grammar.

First of all, let assume that a grammar is in Greibach normal form. Note that any production applied to a derivation adds a single terminal symbol. This means that any derivation of a word of length n has length n (i.e. productions are applied n-times in a derivation). A method of construction of a derivation is simple. Of course, we start with the initial symbols S of the grammar and apply a S-production with right hand part beginning with the first letter of the word. In the next steps, we take the leftmost nonterminal symbol A of an intermediate derivation word and the next consecutive letter a of the word and apply an A-production beginning with the terminal symbol a.

If a grammar is simple, i.e., for every nonterminal symbol A and for every terminal symbol a there is at most one A-production with the right hand side beginning with a, then there is no ambiguity in the choice of productions. Otherwise, when the grammar is not simple, there is a question how to choose a production. We either can make a nondeterministic choice between all A-production with the right hand side starting with given terminal symbol, or can check if any possible derivation produces the given word w. In case of checking all possible derivations, assuming that we have no more than k productions for the choice, we may have up to k^n derivations, where length of the word w is equal to n. This means that computational complexity of this method is exponential, what makes it useless in practice. Note: a concept of nondeterminism will be discussed in further parts of this book.

There are more algorithms for membership test, many of them being applicable to special forms of context-free grammars. We discuss here an algorithm invented by J. Cocke, H. Younger and T. Kasami. The algorithm is called the Cocke-Younger-Kasami algorithm or the CYK algorithm for short. The CYK algorithm operates on a context-free grammar in Chomsky normal form.

The way of determining whether a word w of length n is generated in a grammar G in Chomsky normal form is outlined as follows:

1. split the word w into n substrings of length 1 (every letter of the word w makes up a substring, in this case) and find out nonterminal symbols generating every substring. This operation is a simple lookup for productions of the form $A \to a$, where a is a given substring of length 1,
2. having nonterminal symbols generating substrings of the word w not longer than k, we can find nonterminals generating strings of length $k+1$:
   a. split a substring of length $k+1$ to all possible pairs of substrings (i.e. prefixes of lengths $1, 2, 3, \ldots, k$ and the corresponding suffixes of length $k, k-1, k-3, \ldots, 2, 1$),
   b. for every pair of a prefix and the corresponding suffix sets of nonterminal symbols generating them have already been determined,

c. find out the set of all nonterminals A such that there is a production $A \to BC$, where B and C are nonterminals generating the prefix and the corresponding suffix,
d. nonterminals found out in the point c generate the given string of length $k+1$ and no other nonterminal does,

3. finally, we get the set of all nonterminal symbols generating the substring of length n, i.e., generating the word w. The word w is generated in the grammar if and only if the initial symbol of the grammar is in this set.

Assuming that $w = a_1 a_2 \dots a_n$ is an analyzed word and V_i^j is the set of all nonterminals generating the substring $a_i a_{i+1} \dots a_{i+j-1}$ of the word w, for $j = 1, 2, \dots, n$, $i = 1, 2, \dots, n-j+1$, the Cocke-Younger-Kasami algorithm can be formulated as follows:

```
begin
1. find out all sets V_i^1 of nonterminal symbols
   generating the letter a_i of the word w
2. for consecutive length values j = 2, 3, ..., n
   of substrings of the word w do
3.   for consecutive substrings a_k ... a_{k+(j-1)} =
           = a_1 ... a_{1+(j-1)} a_2 ... a_{2+(j-1)} ..., a_{n-j+1} ... a_{(n-j+1)+(j-1)} do
     begin
4.     initialize sets V_k^j to the empty set
5.     for splits of a_k ... a_{k+(j-1)} to prefix a_k ... a_{k+l-j)}
       and suffix a_{k+l} ... a_{k+(j-1)}, l = 1,2 ... j-1 do
6.       find out all productions A -> BC
         s.t. B ∈ V_k^l and C ∈ V_{k+l}^{j-l}
         and include all such A's into the set V_k^j
     end
7. the word w is generated in the grammar
   if and only if the initial symbol
   of the grammar is included into the set V_1^n
```

Let us analyze the computational complexity of the CYK algorithm. Observe that costs of the following operations are upper-bounded by constants:

- initialization of sets V_i^1 in operation 1,
- getting a substring in operation 3,
- initialization of sets V_k^j to empty sets in operation 4,
- splitting a string into a prefix and a suffix in operation 5,
- finding out productions in operation 6 (checking all production of the grammar, the number of production is fixed, so then cost of this operation is bounded by a constant),
- including left hand sides A of productions into sets.

Finding out productions in operation 6 is a dominant operation of this algorithm. The number of executions of this operation is equal to:

$$\sum_{j=2}^{n}\sum_{k=1}^{n-j+1}(j-1)=\sum_{j=2}^{n}(n-j+1)(j-1)=$$

$$-\sum_{j=2}^{n}j^2+(n+2)\sum_{j=2}^{n}j-(n-1)(n+1)=$$

$$-(\frac{n^3}{3}+\frac{n^2}{2}+\frac{n}{6}-1)+\frac{(n+2)(n+2)(n-1)}{2}-n^2+1=\frac{n^3-n}{6}$$

what means that complexity of the CYK algorithm is of the range $\Theta(n^3)$ with regard to length of an analyzed word.

The basic version of the CYK algorithm is used to find out if a word is generated in the grammar, but it does not allow for finding a derivation tree. A modified version of the CYK algorithm, with the extended version of operation 6, gathers information necessary for building derivation trees:

6. find out all productions $A\to BC$ s.t. $B\in V_k^l$ and $C\in V_{k+l}^{j-l}$
   and include all such A's into the set V_k^j,
   store right hand side BCod the production $A\to BC$
   and parameter l in the set $A_\to$ attached to A
   (left hand side of the production)

The above operation attaches right hand side of every A-production determined by the operation 6 to the nonterminal symbol A. It means that every nonterminal symbol A in every set V_k^j has some associated set $A_\to$ of right hand sides of A-production generating corresponding substring of the word w. The following algorithm generates derivations tree based on results of the extended CYK algorithm. Parameters of the function `generate` (generate a subtree of the derivation tree) denote: `k` - the position in the word of the first letter of the substring, `j` - length of the substring, `A` - the nonterminal generating the substring, `here` - the position of the nonterminal `A` in the tree.

```
generate(k,j,A,here)
begin
 if j=1 then
 begin
   put the symbol A at the place here,
   draw an edge from A down to the symbol
 end else
 begin
   newTree:=false
   currently build tree is the current copy
   for every BC,l∈A→ do
   begin
     if newTree then
       create a copy of the derivation tree
       built before the current call
```

```
      of this function, make newly created copy
      to be the current copy,
    apply subsequent operations to the
      current copy
    put the symbol A at the place here
    draw a left edge to a leftVertex
    call generate(k,l,B, leftVertex)
    draw a right edge to a rightVertex
    call generate(k+l,j-l,C, rightVertex)
    newTree := true
  end
 end
end
```

The first call of the function `generate` requests building the whole derivation tree for an investigated word w of length n and is as follows:
`generate(1,n,S,position-of-the-root).`

2.6 Applications

This section is devoted to selected application of context-free grammars. The section is a roadside of a main discussion on formal languages, automata and computability. Topics included in this section are a small part of a compilers practice. They can be used in an elementary project on parsing basic constructions like arithmetical expressions, which are among the most complex parts of programming languages. This section is not aimed on complete and detailed presentation of parsing. It is rather a signalization of the theme.

2.6.1 Translation grammars

This section is focused on some modifications of context-free grammars. Despite that modifications presented here are not included into main flow of discussion on context-free grammars, the associated practical importance makes them valuable and justifies their presentation in the book. Two extensions of context-free grammars are presented, namely translation grammars and LL(1) grammars. These types of grammars will be used to parse arithmetic expressions and translate them to postfix form.

Translation grammars stem from context-free grammars. They could be seen as context-free grammars with simultaneous derivation of related words.

Definition 2.6. Translation grammar is a context-free grammar $G = (V, T, P, S)$ with the set T of terminal symbols split into two disjoint subsets T' and T'' of primary and secondary symbols, i.e. $T = T' \cup T''$, $T' \cap T'' = \emptyset$.

A translation grammar is a context-free-grammar producing a context-free language $L(G)$ over the alphabet T of terminal symbols. On the other hand, we can say that the translation grammar produces two languages: the primary language $L'(G)$ and the translation language $L''(G)$. The primary language is obtained from the language $L(G)$ by removing translation symbols from its words. Then again, translation language is obtained from the language $L(G)$ by removing primary symbols from its words.

2.6.2 LL(1) grammars

In this section, we assume that, for a context-free grammar $G = (V, T, P, S)$, any word $w \in T^*$ has a special end-of-word symbol $\lhd$ appended. This symbol is neither nonterminal symbol, nor terminal symbol. It is used for marking end of any intermediate and terminal word of a derivation of the word w.

Assume that $p : A \to \alpha$ is a production in a context-free grammar $G = (V, T, P, S)$, where $A \in V$, $\alpha \in \{V \cup T\}^*$. Let us define the following sets of symbols:

- $FIRST(p) = FIRST(\alpha)$ is the set of those terminal symbols, which may open any intermediate word derivable from α.
- $FOLLOW(p) = FOLLOW(A)$ is the set of those terminal symbols and end-of-word symbol, which may directly follow A in any intermediate word derived from the beginning symbols S of the grammar G.
- $SELECT(p) = \begin{cases} FIRST(p) \cup FOLLOW(p) & \text{if } p \text{ is nullable} \\ FIRST(p) & \text{otherwise} \end{cases}$

Definition 2.7. A context-free grammar $G = (V, T, P, S)$ is $LL(1)$ grammar if and only if for every nonterminal symbol $A \in V$ all A-productions have SELECT sets pairwise disjoint. This condition is called LL(1) uniqueness condition.

LL(1) grammars are tools for building top-down membership analyzer (top-down parser). LL(1) grammars are tools for constructing **L**eftmost derivation of input word, processing the word from **L**eft to right. The derivation is constructed based on **1** input symbol at a time.

Note that the uniqueness condition of the LL(1) grammars is similar to the Greibach uniqueness condition. This allows for an easy construction of the derivation of a word: for a given leftmost nonterminal symbol $A \in V$ in an intermediate derivation word and a given input symbol $a \in T$ we apply this A-production to the nonterminal A, which has the terminal a in its SELECT set. When the right-hand side of the applied production begins with the terminal symbol a, the input is shifted to the next input symbol. Translation symbols in an intermediate derivation words are skipped during this processing.

Chapter 3
Context-sensitive and unrestricted grammars

The class of context-sensitive languages follows the class of context-free languages. In the hierarchy of languages, a so called Chomsky hierarchy, the next classes of languages, besides regular languages and context-free languages, are context-sensitive and recursively enumerable languages. Context-sensitive languages are generated by context-sensitive grammars, which happen to be a generalization of context-free grammars. Recursively enumerable languages are generated by unrestricted grammars, which are an extension of context-sensitive grammars. We will also distinguish the class of recursive languages. However, there is no class of grammars generating recursive languages. The class of recursive languages is separated from the class of recursively enumerable based on special class of automata. This topic will be presented in Chapter 4.

As mentioned above, context-sensitive and unrestricted grammars are much more complex than context-free and regular grammars. Likewise, a structure of words of these classes of languages is much more complex than a structure of words of context-free languages. There are no properties showing restriction of words structure or finiteness or regularity of the language structure. Algebraic characterization of these languages in the set of all words is not known. Therefore, we do not have effective tools for processing these languages, as it is in case of context-free languages, e.g., such tool as the pumping lemma is not known for context-sensitive languages.

This Chapter presents a short presentation of the basic properties of context-sensitive and recursively enumerable languages.

3.1 Context-sensitive grammars

Productions of context-sensitive grammars satisfy monotonic condition (also called noncontracting condition). For this reason, context-sensitive grammars are also called monotonic grammars or noncontracting grammars.

Definition 3.1. A grammar $G = \{V, T, P, S\}$ is context-sensitive if and only if its productions are monotonic (or noncontracting), i.e., they are of the form:

$$\alpha \to \beta, \quad \text{where:} \quad \alpha, \beta \in (V \cup T)^* \text{ and } 0 < |\alpha| \leq |\beta| .$$

where $|w|$ denotes a length of the word w.

Definition 3.2. Context-sensitive languages are those generated by context-sensitive grammars, and only those.

The class of context-sensitive grammars and the class of context-sensitive languages are denoted by CSG and CSL, respectively.

The monotonic condition excludes the empty word ε from context-sensitive languages. In terms of the above definition, any language, which includes the empty word, is not context-sensitive. This is the strict meaning of CSL and CSG classes.

But it would be unreasonable to exclude from the CSL class a context-sensitive language with the empty word attached. Thus, any language L, such that $L - \{\varepsilon\}$ is a context-sensitive language, will be included in the CSL class. Note, that having a language L generated a monotonic grammar, we can add the empty word ε to L by attaching the production $S \to \varepsilon$ to the grammar, where S is the initial symbol of the grammar. This production breaks monotonicity of the grammar, so it will be considered to be the unique exception of a context-sensitive grammar. This meaning of context-sensitivity is called extensive context-sensitivity.

Summarizing the above notes, the CSL and CSG classes will be considered in the strict or extended sense depending on a context of discussion. In the sequel, we will not distinguish between strictness and extensiveness of context-sensitivity, if this does not lead to confusion.

Definition 3.3. A grammar $G = (V, T, P, S)$ is in context-sensitive normal form if and only if its productions have the following form:

$$\gamma A \delta \to \gamma \alpha \delta , \quad \text{where} \quad A \in V, \ \alpha, \gamma, \delta \in (V \cup T)^*, \ \alpha \neq \varepsilon .$$

γ and δ are called left and right context, respectively and $A \to \alpha$ is called the core of the production.

Note that a grammar in a context-sensitive normal form is a context sensitive grammar, since its productions are monotonic. Then the class of grammars in context-sensitive normal form is included in the CSG class. The question is whether the CSG class is equivalent to the class of grammars in context-sensitive normal form. This question is equivalent to the question if for any context-sensitive grammar we can find a grammar in context-sensitive normal form. The answer is affirmative. Hence, grammars in context-sensitive normal form do not create a new class of grammars. Therefore, we will simply refer to context-sensitive grammars in normal form.

Note that the core is simply a context-free production. Also observe, that only core of a production can affect derivation. However, the core of a production can

be used only if left and right context are preserved. This is why grammars with monotonic productions are called context-sensitive grammars.

Now, let us justify that every context-sensitive grammar can be transformed to normal form:

Lemma 3.1. *Any context-sensitive grammar $G = (V, T, P, S)$ can be transformed to an equivalent grammar in normal form (both grammars generate the same language).*

Proof. We construct a context-sensitive grammar in normal form which is equivalent to the grammar G. A detailed proof is left to the reader.

The grammar in normal form equivalent to a given context-sensitive grammar is constructed as follows:

1. for every terminal symbol $a \in T$
   a. create a new non-terminal symbol A_a and convert every production of the grammar G replacing every occurrence of the symbol a with the new non-terminal A_a,
   b. add the new production $A_a \to a$.

   We get a new grammar $G_T = (V \cup V_T, T, P' \cup P_T, S)$ with an extended set $V \cup V_T$ of nonterminal symbols and an extended set $P' \cup P_T$ of productions, where $V_T = \{A_a : a \in T\}$, P' is the set of productions converted from P, $P_T = \{A_a \to a\, a \in T\}$. Productions of P' have now a form $A_1 A_2 \ldots A_k \to B_1 B_2 \ldots B_l$, where $k \leq l$ (since the grammar G is a context-sensitive, i.e. it is a monotonic grammar) and all symbols in the production are nonterminal symbols, i.e. $A_1, \ldots, A_k, B_1, \ldots, B_l \in V \cup V_T$,
2. let us split the set P' of productions to subsets P'_n (corresponding to the subset P_n of P) of productions in normal form and P'_m (corresponding to the subset P_m of P) of productions that are monotonic, but not in normal form, $P = P_n \cup P_m$ and $P' = P'_n \cup P'_m$. Let us enumerate productions of the set P'_m, which are not in normal form,
3. for every production $r : A_1 A_2 \ldots A_k \to B_1 B_2 \ldots B_l$ of the set P'_m with assigned number r do:
   a. create a new nonterminal symbols A^r_k,
   b. replace the production $r : A_1 A_2 \ldots A_k \to B_1 B_2 \ldots B_l$ with the following set r_N of $k+1$ productions in normal form:
      - $A_1 \ldots A_{k-1} A_k \to A_1 \ldots A_{k-1} A^r_k$, where $\gamma = A_1 \ldots A_{k-1}$ is the left context, the right context is empty and $A_k \to A^r_k$ is the core,
      - $A_1 A_2 \ldots A_{k-1} A^r_k \to B_1 A_2 \ldots A_{k-1} A^r_k$, where the left context is empty, $\delta = A_2 A_3 \ldots A_{k-1} A^r_k$ is the right context and $A_1 \to B_1$ is the core,
      - $B_1 A_2 A_3 \ldots A_{k-1} A^r_k \to B_1 B_2 A_3 \ldots A_{k-1} A^r_k$, where $\gamma = A_1$ is the left context, $\delta = A_3 \ldots A_{k-1} A^r_k$ is the right context and $A_1 \to B_1$ is the core,
      - $\ldots$
      - $B_1 \ldots B_{k-2} A_{k-1} A^r_k \to B_1 \ldots B_{k-2} B_{k-1} A^r_k$, where $\gamma = B_1 \ldots B_{k-2}$ is the left context, $\delta = A^r_k$ is the right context and $A_{k-1} \to B_{k-1}$ is the core,

- $B_1 \dots B_{k-1} A_k^r \to B_a \dots B_{k-1} B_k \dots B_l$, where $\gamma = B_1 \dots B_{k-1}$ is the left context, the right context is empty and $A_k^r \to B_k \dots B_l$ is the core,

4. finally we come up with the following context-sensitive grammar in normal form $G_N - (V_N, T, P_N, S)$, where:
   - $V_N = V \cup V_T \cup \bigcup_r \{A_{k_r}^r\}$ and r is the number of a production $A_1 A_2 \dots A_k \to B_1 B_2 \dots B_l$ from P_m',
   - $P_N = P_T \cup P_n \cup \bigcup_r R_N$, where R_N is the set of productions in normal form corresponding to the production r of P_m.

3.2 Unrestricted grammars

The class of unrestricted grammars is the more general class of grammars. Unrestricted grammars are similar to context-sensitive grammars except that productions are not required to be monotonic (noncontracting). Unrestricted grammars generate the class of recursively enumerable languages, which will be denoted as REL class of languages. The formal definitions are as follows:

Definition 3.4. A grammar $G = \{V, T, P, S\}$ is unrestricted if and only if its productions are of the form:

$$\alpha \to \beta, \quad \text{where: } \alpha, \beta \in (V \cup T)^* \text{ and } 0 < |\alpha|$$

where: $|w|$ denotes length of the word w.

Definition 3.5. Recursively enumerable languages are those generated by unrestricted grammars, and only those.

Context-sensitive grammars are special cases of unrestricted grammars. Thus, the class of context-sensitive languages is a subclass of recursively enumerable languages, i.e. $CSL \subset REL$. But it is not obvious if this inclusion is proper, i.e. if $CSL \neq REL$ because nearly every language that we can imagine is context-sensitive. In the consecutive Chapters we will construct languages that are recursively enumerable, but not context-sensitive.

Chapter 4
Turing machines

In previous chapters methods of generating languages were studied. Those methods are based on different types of grammars and on regular expressions. This and next chapters are devoted to a discussion on methods of acceptation of languages. Acceptation of languages is based on different types of automata: Turing machines, linear automata, push-down automata and finite automata. Identification of grammars with generation of languages and automata with acceptation of languages is a subjective and intuitive categorization done by authors. However, it reflects nature of tools for processing languages.

Turing machines (and other types of automata) can be interpreted as models of computation. Turing machines is a universal model of computation that is used for such purposes as, for instance, acceptation of languages, computing functions, solving problems.

In this Chapter Turing machines, and automata in general, will be employed as tools of acceptation of languages, i.e. they will be queried whether given word is in the language accepted by a given automaton or not.

Turing machines can also compute functions. Such machines compute functions with natural numbers as domain and co-domain. Another type of Turing machines solves problems like, for instance, the sorting problem, the shortest paths problem, etc.

4.1 Deterministic Turing machines

In this book two categories of Turing machines (automata, in general) will be studied: deterministic and nondeterministic. Roughly speaking, computation of an automaton is a sequence of configurations organized according to some control information. An automaton is a deterministic one if and only if there is at most one possibility of doing a transition in any configuration. If, for given automaton, there is a choice of doing a transition in some configuration, then such an automaton is a nondeterministic one.

In this section different categories of deterministic Turing machines are studied: a basic model, a model with guard, a multi-track model and a multi-tape model. At the end of this Chapter nondeterministic Turing machines are discussed. Equivalence of these categories of Turing machines is drawn, i.e. it is shown that for a Turing machine in any model, we can find an equivalent machine in any other model. Equivalence of Turing machines (equivalence of automata, in general) means that they accept the same language, compute the same function or solve the same problem. The discussion leads to the main goal of this Chapter: equivalence of deterministic and nondeterministic Turing machines.

4.1.1 Basic model of Turing machines

A definition of the basic model of deterministic Turing machine is given below. Later in this chapter other deterministic models of Turing machines are discussed. They are proved to be equivalent with the basic model. As it was stated above, equivalence with regard to accepted languages is considered. However, generalization of equivalence to Turing machines computing functions or solving problems is straightforward.

Definition 4.1. A Turing machine in basic model is a system

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F, C)$$

with components as follows:

Q – a finite set of states,
Γ – a finite set of tape symbols (tape alphabet),
B – the blank symbol (of tape alphabet), $B \in \Gamma$,
Σ – an input alphabet, $\Sigma \subset (\Gamma - \{B\})$,
q_0 – the initial state, $q_0 \in Q$,
F – a set of accepting states, $F \subset Q$,
C – a condition, its satisfaction is necessary and sufficient to stop computation,
δ – a transition function, which is a mapping:
$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$

where L, R denote left and right directions.

A Turing machine could be interpreted as a physical mechanism shown in Figure 7.1. This mechanism consists of:

- a control unit, it is in a state of Q,
- a one way infinite tape split in cells, every cell contain a symbol (exactly one) of tape alphabet Γ,
- a head, it is placed over a cell of a tape, it reads a symbol hold in a cell, it stores desired symbol in the cell, it shifts left or right.

Turing machines do computation for given input data. A computation of a given Turing machine is done according to the following intuitive procedure:

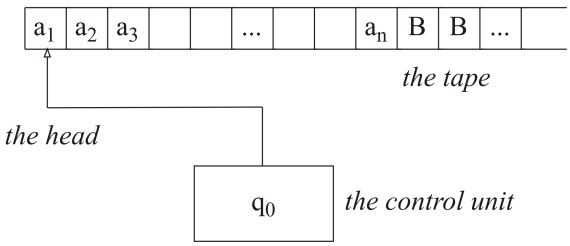

Fig. 4.1 Basic model of Turing machine.

1. the initial configuration of a given Turing machine is described as follows:
   a. input data, a word $w = a_1 a_2 \ldots a_n$ over input alphabet Σ, is stored in n beginning cells of the tape, c.f. Figure 7.1,
   b. all other cells of the tape, which is infinite to the right, are filled in with the blank symbol B,
   c. the head of the Turing machine is placed over the first (leftmost) cell of the tape,
   d. the control unit is in the beginning state q_0,
2. if the stop condition C is satisfied, then computation is halted. The configuration is called the final configuration. The machine responses whether its control unit is in an accepting state or not,
3. if the stop condition C is not satisfied, then - based on the state q of the control unit and on the symbol X read by the head - the Turing machine is doing the following actions:
   a. the value (p, Y, D) of the transition function $\delta(q, X)$ is computed,
   b. the head stores the tape symbol Y in the cell under it,
   c. the control unit switches to the state p,
   d. the head shifts one cell in the direction D,
4. computation goes to the point 2.

The above intuitive procedure could be adapted to Turing machines, which compute functions or solve problems. This adaptation needs a redefinition of input data. Input data of a Turing machine is:

- a sequence of arguments of the function computed by the machine. Arguments of a function are encoded as numbers e.g. in binary or decimal positional system or in unary system. Of course, some sort of separators between arguments must be used,
- a data defining an instance of the problem solved by the machine. This data is encoded in a way depending on type of data.

Note that the stop condition C can never be satisfied and the machine will be doing an infinite computation. When infinite computation is done, the input data is

not accepted by the machine, i.e. the input word does not belong to the language accepted by the Turing machine. It should be underlined that a Turing machine always stops its computation in an accepting state when the input data of this machine is a word of the language accepted by the machine. Turing machines raise a difficult problem: when a one is doing long computation, there is no assertion if the machine has fallen into infinite computation or it will stop computation in future.

Remark 4.1. Turing machines will be designed assuming that, if a machine terminates computation, its head is placed over the first (leftmost) cell and:

- when the machine accepts a language, then all cells of the taped are filled in with the blank symbol B,
- when the machine computes a function, then the value of the function is stored in beginning cells of the. All other cells are filled in with the blank symbol B . The stored value is the correct result if and only if the machine stopped computation in an accepting state,
- when the machine solves a problem, then output data is stored in beginning cells of the tape. All other cells of the tape are filled in with the blank symbol B. Output data is a correct solution of a computed instance of the problem if and only if the machine stops computation in an accepting state.

Assumptions of the above remark are not included into Definition 4.1, but they are wished for clarity of computation. An epilogue of a computation guarantying satisfaction of the above assumptions is called a cleaning procedure.

Definition 4.2. A step description (a configuration) of a Turing machine

$$M = (Q, \Gamma, \Sigma, \delta, q_0, B, F, C)$$

is a sequence of symbols:

$$\alpha_1 \, q \, \alpha_2$$

where:

- $q \in Q$ is the current state of the control unit of a Turing machine,
- α_1 is a sequence of symbols stored in cells beginning from the leftmost one and ending with the cell prior the one under the head,
- α_2 is a sequence of symbols stored in cells beginning from the one under the head, going to the right and ending with the rightmost one holding a non-blank symbol.

Note that both α_1 and α_2 sequences of symbols are words over the tape alphabet Γ and that any of these sequences may be the empty word. However, none of these two sequences can be infinite. This is due to the following reasons:

- an input data is finite, so then only a finite number of cells are filled in with non-blank symbols in the input configuration,

- after any step of computation only finite number of cells could be visited by the head.

For instance, the initial step description (configuration) is of the form $q_0\,w$, where q_0 is the initial state, w is the input data. In this case α_1 is the empty word. On the other hand, a step description $\alpha_1\,q$ informs that all cells from the one under the head to the right are filled in with the blank symbol B. Now, α_2 is the empty word. Finally, when a Turing machine accepting a language ends its computation in a state q, then - according to Remark 4.1 - q will be the final step description.

Let us analyze transitions done by Turing machines. We assume that a step description is characterized by the following sequence of symbols:

$$X_1\,X_2\,\ldots\,X_{i-1}\,q\,X_i\,\ldots\,X_n$$

Recall that:

- q is the current state of the control unit,
- $X_1\,X_2\,\ldots\,X_{i-1}$ is the sequence of symbols of the tape alphabet Γ stored in cells preceding the cell under the head,
- $X_i\,X_{i+1}\,\ldots\,X_n$ is the sequence of symbols stored in cells beginning from the one under the head, going to the right and ending with the rightmost one containing non-blank symbol,
- the head is placed over the cell with the X_i symbol stored in. If the sequence $X_i\,X_{i+1}\,\ldots\,X_n$ is the empty word, then the head reads the blank symbol B.

The next step description is determined by the transition function:

- if the value of the transition function is $\delta(q,X_i)=(p,Y,R)$, i.e. the control unit switches to the state p, the head stores Y and shifts right, then we get the following configuration,
$$X_1\,X_2\,\ldots\,X_{i-1}\,Y\,p\,X_{i+1}\,\ldots\,X_n$$
- if the value of the transition function is $\delta(q,X_i)=(p,Y,L)$, then we get the following configuration:

$$X_1\,X_2\,\ldots\,X_{i-2}\,p\,X_{i-1}\,Y\,X_{i+1}\,\ldots\,X_n$$

We will use the symbol $\succ$ to denote a transition of a Turing machine. The transition symbol $\succ$ may be supplement with a Turing machine name $\succ_M$ to emphasize that a transition concerns a given Turing machine. It also can be supplemented with a superscript $\succ^k$ to notify k transitions done.

The above two transitions done by a Turing machine will be denoted as follows:

$$X_1\,X_2\,\ldots\,X_{i-1}\,q\,X_i\,\ldots\,X_n \succ X_1\,X_2\,\ldots\,X_{i-1}\,Y\,p\,X_{i+1}\,\ldots\,X_n$$

$$X_1\,X_2\,\ldots\,X_{i-1}\,q\,X_i\,\ldots\,X_n \succ X_1\,X_2\,\ldots\,X_{i-2}\,p\,X_{i-1}\,Y\,X_{i+1}\,\ldots\,X_n$$

Definition 4.3. Transitions of a Turing machine create a binary relation in the space of all possible configurations of the machine, i.e. any two configurations are related

if and only if the second one is derived from the first one utilizing the transition function. This relation is called the transition relation of a given Turing machine. The transitive closure of the transition relation is denoted $\succ^*$.

Definition 4.4. A computation of a Turing machine $M = (Q, \Gamma, \Sigma, \delta, q_0, B, F, C)$ is a sequence of configurations $\eta_1, \eta_2, \ldots, \eta_n$ such that η_1 is the beginning configuration, η_n is the final configuration and a pair of any two successive configurations belongs to the transition relation. If the machine has fallen into an infinite computation, then its computation is an infinite sequence of configurations $\eta_1, \eta_2, \eta_3 \ldots$ such that η_1 is the beginning configuration and any pair of two successive configurations belongs to the transition relation. A finite computations is denoted as $\eta_1 \succ \eta_2 \succ \ldots \succ \eta_n$ and infinite computation is denoted as $\eta_1 \succ \eta_2 \succ \eta_3 \succ \ldots$.

Now we give a formal definition of acceptation of an input by a Turing machine:

Definition 4.5. A Turing machine accepts its input if and only if the computation terminates in an accepting state. In other words, a Turing machine accepts its input if and only if the pair of the initial configuration and the final configuration belongs to transitive closure of the transitive relation i.e. $\eta_1 \succ^* \eta_T$, where η_1 is an initial configuration and η_T is a final configuration.

Based on the above discussion we give now formal definitions of some concepts.

Definition 4.6. The language $L(M)$ accepted by a Turing machine M is the set of words $w \in \Sigma^*$ accepted by the Turing machine.

Definition 4.7. A function computed by a Turing machine M is a mapping from a space of input data into a space of output data. If the machine accepts its input, then its output is the correct value of the function. Otherwise, when the machine stops computation, but not accepts the input, or if it is doing infinite computation, the function is undefined for such input data.

Remark 4.2. We assume that the blank symbols B will neither separate non-blank symbol on tape, nor be placed in leftmost cells prior to a non-blank symbol. This assumption is not required by definitions and concepts discussed so far. However, it simplifies designing of Turing machines for given tasks.

4.1.2 Turing machine with the stop property

As noticed before, some Turing machines can fall into an infinite computation. Some other will terminate their computation for any input data. This observation draws the definition of the subclass of Turing machines which always terminate their computation.

Definition 4.8. A Turing machine in basic model with the stop property is a system introduced in Definition 4.1

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F, C)$$

and such that it terminates its computation for any input data.

It is worth to underline that a function or a language computed by a Turing machine with the stop property can also be computed by a Turing machine without stop property. Moreover, such the Turing machine without stop property can perform infinite computation for some input data.

Two Turing machines, one with the stop property and another one without the stop property, if compute the same function or accept the same language, must terminate computation in accepting states for the same input data. Both machines may also terminate computation in rejecting states for the same data. They may yield different outputs only for such input data, which is not accepted. In such the case, the machine with the stop property terminates its computation in a rejecting state and the second machine falls in infinite computation.

Definition 4.9. Turing machine are considered to be equivalent if and only if they terminate computation in accepting state for the same input data.

4.1.3 Simplifying the stop condition

Now we will slightly change definitions of Turing machines in basic model in order to simplify definition of termination of its computation.

Definition 4.10. Turing machine with the halting accepting state is a system:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

where:

- $F = \{q_A\}$ - there is only one accepting state q_A,
- the stop condition is satisfied if and only if the machine switches to the accepting state q_A,
- other components of the system are as described in Definition 4.1.

Proposition 4.1. *Turing machines with the halting accepting state are equivalent to Turing machines in basic model.*

Proof. First of all, a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, \{q_A\})$, with halting accepting state will formally match Definition 1, when the stop condition is included in the system, i.e. $M = (Q, \Sigma, \Gamma, \delta, q_0, B, \{q_A\}, C)$, where C is satisfied if and only if the machine switches to state q_A.

On the other hand, a Turing machine in basic model $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F, C)$ can be updated to a machine with halting accepting state by:

- adding new states $q_\#$ and q_A,
- doing two transitions: $(q_\#, X, R)$ (q_A, Y, L), when the stop condition of the Turing machine in basic model is satisfied and the machine is in accepting state, where X and Y are symbols previously stored in cells under the head. These two transitions just switch the machine to the new state q_A, keeping contents of the tape unaffected and places the head in the same position as before these transitions,
- replacing former accepting states by q_A,
- redefining the stop condition: computation is halted if and only if the machine switches to the state q_A (now the only accepting state).

The machine with halting accepting state may fall into infinite computation when the machine in basic model stops its computation in non-accepting state. Anyway, the machine with the halting accepting state terminates its computation in accepting state if and only if the machine in basic model does the same.

This proves equivalence of both machines with regard to accepted language. Therefore, Turing machines in basic model are equivalent to Turing machine with halting accepting state.

Definition 4.11. Turing machine with halting states is a system

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F, R)$$

such that it terminates its computation for any input data, where:

- $F = \{q_A\}$ - includes only one accepting state q_A,
- $R = \{q_R\}$ - includes a special non-accepting state q_R,
- computation for any input always reaches one of states q_A or q_R,
- computation stops if and only if it reaches q_A or q_R,
- other components of the system are as described in Definition 4.1.

Proposition 4.2. *Turing machines with halting states are equivalent to Turing machines in basic model with the stop property.*

Proof. Just modify the proof of Proposition 4.1 to justify this Proposition.

4.1.4 Guarding the tape beginning

Basic model of Turing machines raises a practical problem how to detect the tape beginning. Of course, a general solution of this problem is just to store special symbols in the first cell of the tape, which replace symbols defined by transition function. However, such a solution enlarges the set of states, the tape alphabet and the transition function. A model of Turing machine with guard avoids this problem.

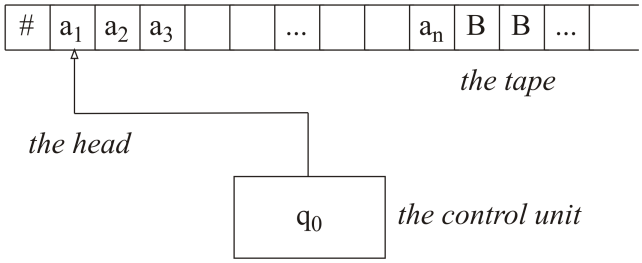

Fig. 4.2 Turing machine in basic model with guard.

Definition 4.12. Turing machine in basic model with guard is a system

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, \#, F, C)$$

where:

- # - the guard symbol (of tape alphabet), $\# \in \Gamma$, $\# \notin \Sigma$,
- other components are as in Definition 4.1,
- the input configuration is shown in Figure 4.2,
- the head can visit the first cell (with guard), but cannot change its contents.

Proposition 4.3. *Turing machines in basic mode with guard are equivalent to Turing machines in basic model.*

Proof. Given a Turing machine in basic model

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F, C)$$

its equivalence with guard could be constructed by supplementing the system with the guard symbol, storing the guard symbol in the leftmost cell, storing input data from the second leftmost cell and placing the head over the second leftmost cell (over the leftmost input symbol)

$$M_G = (Q, \Sigma, \Gamma, \delta, q_0, B, \#, F, C)$$

Other components of the M_G stay unchanged. Any computation of M_G will be exactly the same as for M and the head will never visit the guard cell.

And oppositely. Given a Turing machine in basic model with guard

$$M_G = (Q, \Sigma, \Gamma, \delta, q_0, B, \#, F, C)$$

the following machine is equivalent

$$M' = (Q', \Sigma, \Gamma, \delta', q_0', B, F', C')$$

The machine M will do the following computation:

- shifts input data one cell right, stores the guard symbol # in the first cell and leaves the head at the first input symbol (on the second leftmost cell),
- simulates computation of M_G,
- shifts the output data one cell left (this deletes the guard symbol in the leftmost cell) and leaves the head at the leftmost output symbol (at the leftmost cell).

Remark 4.3. The other models of Turing machines, e.g. Turing machines with the stop property or Turing machines with halting states, can be turned to models with guard. Formulation and proof of equivalence of models with guard and other models of Turing machines are analogous to the proof of Proposition 4.3.

4.1.5 Turing machines with a multi-track tape

A Turing machine with a multi-track tape has the tape split at its length to a given number of tracks. Every track is split into cells and is one way infinite. The head reads symbols of all cells of the same slice (column) and shifts simultaneously over all tracks. An initial configuration of a Turing machine k-tracks tape is shown in Figure 4.3. Input data is stored in beginning cells of the track no 1 while all other cells of the track no 1 and all cells of other tracks are filled in with the blank symbol B.

Definition 4.13. Turing machine in basic model with a multi-track (k-tracks) tape is a system

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F, C)$$

where:

- $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}$ is the transition function
- input data $w = a_1 a_2 \dots a_n$ is represented as sequence of k-tuples $((a_1, B, \dots, B), (a_2, B, \dots, B), \dots, (a_n, B, \dots, B))$, every k-tuple fills in one slice of the tape, c.f. Figure 4.3.
- output data is represented in a way similar to representation of input data, i.e. is stored in beginning cells of the first track while all other cells are filled in with the blank symbol,
- other components are as in Definition 4.1.

Proposition 4.4. *Turing machines with a multi-track tape are equivalent to Turing machines in basic model.*

Proof. First of all, a Turing machine in basic model is a special case of a Turing machine with a multi-track tape having just one track.

Secondly, a Turing machine with k-tracks tape:

$$M' = (Q, \Sigma', \Gamma', \delta, q_0, B', F, C)$$

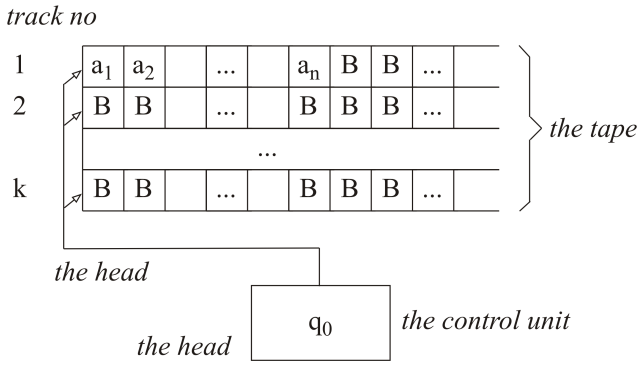

Fig. 4.3 Turing machine in basic model with a multi-track tape.

is equivalent to the following Turing machine in basic model:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F, C)$$

where:

- $\Sigma = \Sigma' \times \{B\} \times \{B\} \times \ldots \times \{B\} -$ *product of k sets*
- $\Gamma = \Gamma' \times \Gamma' \times \ldots \times \Gamma$ *product of k sets*
- $B = (B', B', \ldots, B') -$ *k − tuple*
- other components are as in machine M'.

The above conclusion is not surprising. Turing machines with a multi-track tape work in a way quite similar to Turing machines in basic model, i.e. they have one-way infinite tape, their one head reads and writes data of the whole slice at a time, etc. Therefore, when data of a slice is interpreted as one symbol of a tape alphabet, a Turing machine with a multi-track tape is quite similar to a Turing machine in basic model. Turing machines with a multi-track tape are essentially useful in proofs of equivalence of other, more important, models of Turing machines. This is the main motivation for discussing this model.

4.1.6 Turing machines with two-way infinite tape

Variations of Turing machines discussed so far are slightly modified Turing machines in basic model. Two-way infinite tape is the first important modification of Turing machines in basic model, c.f. Figure 4.4. When Remark 4.2 is employed, Turing machines with two-way infinite tape permit avoiding problems with passing head to beginning of tape or to beginning of data stored on tape: the first cell with the blank symbol prior to nonblank symbols indicates the beginning of data stored on the tape.

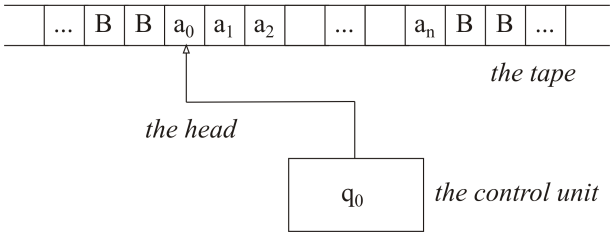

Fig. 4.4 Turing machine with two-way infinite tape.

Definition of Turing machine with two-way infinite tape is identical with Definition 4.1. Turing machine with two-way infinite tape does not need to identify the beginning of tape. However, definition of configuration (step description) of this type of machines is slightly different than that of machines in basic model. The difference is in description of the left sequence of symbols.

Definition 4.14. A step description (a configuration) of a Turing machine with two-way infinite tape $M = (Q, \Gamma, \Sigma, \delta, q_0, B, F, C)$ is the following sequence of symbols:

$$\alpha_1 \; q \; \alpha_2$$

where:

- $q \in Q$ and α_2 are the same as in Definition 4.2,
- α_1 is the sequence of symbols stored in cells left of the head starting with a leftmost non-blank symbol and ending with symbol in the cell prior the one under the head.

Proposition 4.5. *Turing machines with two-way infinite tape are equivalent to Turing machines in basic model.*

Proof. We prove that Turing machines with two-way infinite tape and Turing machines with a multi-track tape are equivalent. Because Turing machines with a multi-track tape are equivalent to Turing machines in basic model, c.f. Proposition 4.13, then we get equivalence declared in this Proposition.

For a given Turing machine in basic model M_1, an equivalent Turing machine with two-way infinite tape M_2 is equal to M_1. Computation of M_1 is being done on the *right half* of tape (the right part of the tape which begins with the cell holding the first symbol of input data).

Assuming that a machine with two-way infinite tape is given as follows:

$$M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q_0^2, B_2, F, C)$$

we will construct a machine in basic model:

$$M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_0^1, B_1, F, C)$$

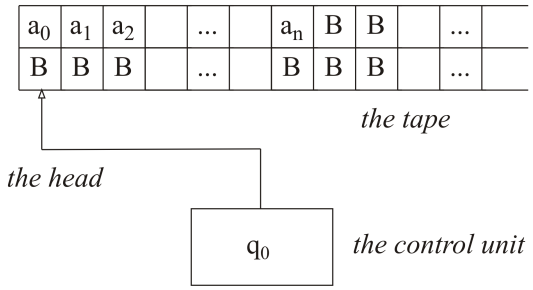

Fig. 4.5 Turing machine with two-track tape - simulation of two-way infinite tape.

First of all, a method of representation of a two way infinite tape must be found out. A one-way infinite tape is a 2-tracks tape. The *right half* of the two-way infinite tape matches to the upper track and the *left half* rotated by 180° matches to the lower track, c.f. Figure 4.4 and Figure 4.5

Computation of M_2 done on the *right half* of tape is followed on the upper track of M_1. Computation of M_2 done on the *left half* of tape is followed on the lower track of M_1 with the head shifting oppositely than the head of M_2. Note that the first cell of the lower track plays a role of guard while content of the *left half* of tape is stored beginning with the second cell, c.f. Figure 4.6.

Formal description of M_1 simulating M_2 is as follows:

- $Q_1 = Q_2 \times \{U,B\} \cup \{q_0^1\}$
- $\Gamma_1 = \Gamma_2 \times \Gamma_2 \cup \Gamma_2 \times \{\not\epsilon\}$
- $\Sigma_1 = \Sigma_2 \times \{B_2\}$
- $F_1 = \{(q,U),(q,L) : q \in F_2\}$
- $B_1 = (B_2,B_2)$

Symbols U and L depict placement of the head of M_2 on the *right* and the *left half* of the tape.

The first transition of M_1 stores guard symbol in the first cell of the lower track and simulates the first transition of M_2 going to either the upper, or the lower track:

$$\delta_1(q_0^1,(a_0,B)) = \begin{cases} ((p,U),(Y,\not\epsilon),R) & if \ \ \delta_2(q_0^2,a_0) = (p,Y,R) \\ ((p,L),(Y,\not\epsilon),R) & if \ \ \delta_2(q_0^2,a_0) = (p,Y,L) \end{cases}$$

When the head of M_1 is placed right of the first cell (computation of M_2 cannot change current *half* of the tape) then:

$$\left.\begin{array}{l} \delta_1\ ((q,U),(X,Z)) = ((p,U),(Y,Z),A) \\ \delta_1\ ((q,L),(Z,X)) = ((p,L),(Z,Y),\overline{A}) \end{array}\right\} \ if \ \ \delta_2\,(q,X) = (p,Y,A)$$

where: A is a direction of the head shift, $\overline{A}$ is opposite to A.

When the head of M_1 is placed at the first cell (computation of M_2 may change current *half* of the tape) then:

$$\left.\begin{array}{l} \delta_1\ ((q,U),(X,\not\epsilon)) = ((p,U),(Y,\not\epsilon),R) \\ \delta_1\ ((q,L),(X,\not\epsilon)) = ((p,U),(Y,\not\epsilon),R) \end{array}\right\} \ if \ \ \delta_2\,(q,X) = (p,Y,R)$$

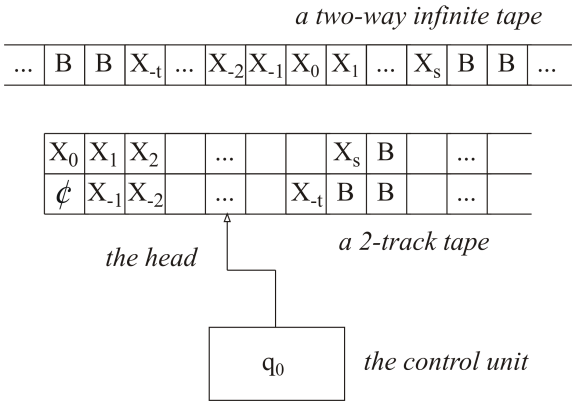

Fig. 4.6 Turing machine with two-track tape - simulation of computation of Turing machine with two-way infinite tape.

$$\left.\begin{array}{l}\delta_1((q,U),(X,¢)) = ((p,L),(Y,¢),R)\\ \delta_1((q,L),(X,¢)) = ((p,L),(Y,¢),R)\end{array}\right\} \; if \;\; \delta_2(q,X) = (p,Y,L)$$

where,

- U and L, coming in states labels, stand for *upper* and *lower* track,
- L and R, coming as the third element of the transition function (3-tuple) value, stand for *left* and *right* direction of head shifts.

In light of Proposition 4.4 and Proposition 4.5 the following conclusion is fairly obvious:

Proposition 4.6. *Turing machines with a multi-track two-way infinite tape are equivalent to Turing machines in basic model.*

4.1.7 Multi-tape Turing machines

A multi-tape Turing machine, c.f. Figure 4.7, satisfies the assumptions:

- it has several tapes and one head for every tape,
- tapes are two-way infinite,
- the initial configuration assumes:
  - the control unit is in the initial state,
  - input data is stored on the first tape,
  - the head of the first tape is placed over the first (leftmost) symbol of input data,
  - all other cells of the first tape and all cells of other tapes are filled in with the blank symbol,
- a transition of a multi-tape Turing machine is as follows:

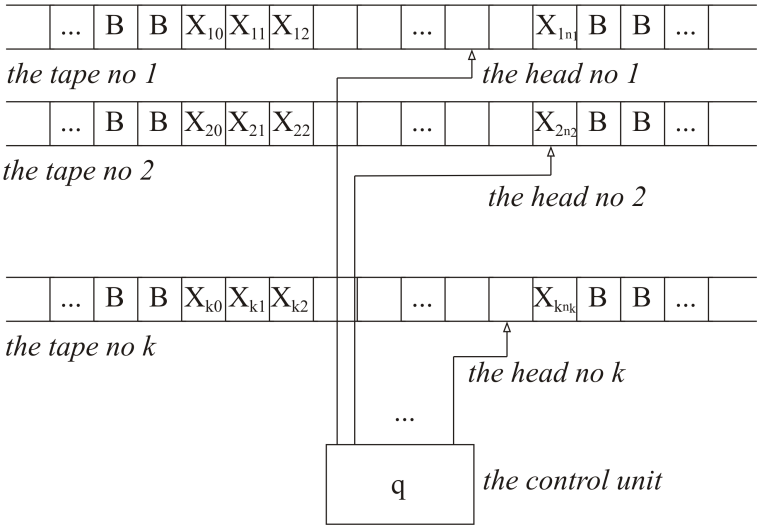

Fig. 4.7 A multi-tape Turing machine.

- the control unit switches to some state,
- every head stores a tape symbols in its cell,
- every head shifts left, right or stays in current position independently on other heads.

Formal definition of a multi-tape Turing machine is as follows:

Definition 4.15. A multi-tape Turing machine (with k-tapes) is a system

$$M = (Q, \Sigma, \Gamma_1 \times \Gamma_2 \times \ldots \times \Gamma_k, \delta, q_0, B, F, C)$$

where:

- $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$ are alphabets of tapes. It is assumed that all tapes have the same alphabet Γ, if not stated differently,
- $\delta : Q \times (\Gamma_1 \times \Gamma_2 \times \ldots \times \Gamma_k) \to Q \times (\Gamma_1 \times \Gamma_2 \times \ldots \times \Gamma_k) \times \{L, R, S\}^k$ is the transition function with directions of head shift: left, right and stop (no shift of a head),
- other components are as in Definition 4.1.

Proposition 4.7. *A multi-tape Turing machines are equivalent to Turing machines in basic model.*

Proof. A Turing machine in basic model is equivalent to some Turing machine with two-way infinite tape, c.f. Proposition 4.5. Moreover, a Turing machine with two-way infinite tape is a case of a multi-tape Turing machine; it is just a multi-tape Turing machine with one tape. Therefore, for any Turing machine in basic model we can directly find an equivalent multi-tape Turing machine.

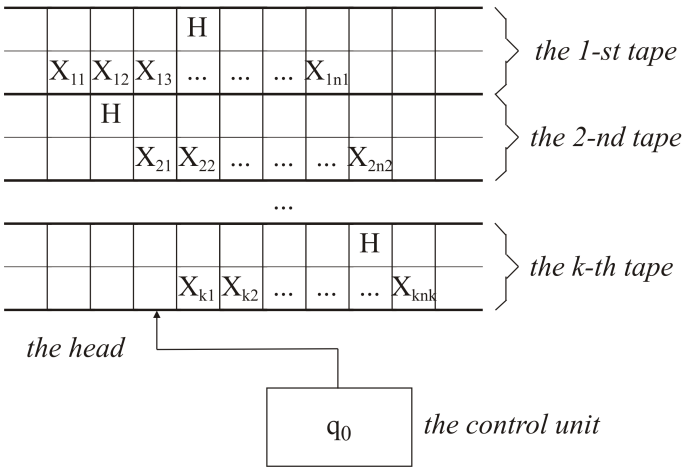

Fig. 4.8 Turing machine with a multi-track two-ways infinite tape simulating a multi-tape Turing machine. The blank symbol is not printed for the sake of clarity, i.e. all cells, which are empty in this Figure, hold the blank symbol.

Now, we give an idea of construction of a Turing machine with a multi-track two-way infinite tape M_1 for a given k-tapes Turing machine M_m:

- k-tapes are represented on a 2-k-tracks two-ways infinite tape, every tape corresponds to a pair of tracks, c.f. Figure 4.8
  - contents of every tape is stored on bottom track of the corresponding pair,
  - the special symbol H stored in a cell of upper track of the corresponding pair marks the head position of the tape of M_m,
  - all other cells of both tracks of the pair are filled in with the empty symbol B,
- initial configuration of M_1 is as follows:
  - the initial state of M_m is stored in the relevant state of M_1,
  - input data stored on the first tape of a multi-tape Turing machine is represented on the lower track of the first pair of tracks of M_1,
  - the head markers of all tapes of M_m are placed in the tape slice holding the first symbol of input data,
- a transition of the machine M_m is simulated by several transitions of the machine M_1:
  - the head of M_1 passes from the tape slice holding the leftmost head marker H to the tape slice holding the rightmost head marker. This is so called *collect-data* pass. Information about symbols under all tape markers (under heads of M_m) is collected during this pass and remembered in a relevant state of M_1,
  - the transition function of M_m is applied to data collected (state of M_m and symbols under heads of M_m). The result of transition function is remembered

in a state of M_1. Recall that transition function of M_m returns: a state, symbols to be stored by heads, directions of heads shifts,
– the head of M_1 is passed to the time slice holding the leftmost tape marker H. This is so called *update* pass. Symbols under head markers and head markers' positions are updated during this pass. Updates are done according to the value of transition function obtained in the previous point.

- computation of M_1 is terminated if and only if the stop condition of M_m is satisfied. Input data of M_1 is accepted if and only if M_m terminates computation in an accepting state.

Note that simulation of a multi-tape Turing machines by Turing machines with one tape requires a huge set of states, what makes transition function highly enlarged. It also increases number of transitions for the same input data.

As it is stated in Proposition 4.7, data collected during left to right pass of the head on simulating machine are stored in the set of states. So, a normal set of states Q_1 of the simulating machine should be extended by Cartesian product of:

- a pass direction: $Q_1 \times \{C,U\}$ to indicate if this is *collect-data* pass or *update* pass,
- the set of states of a multi-tape machine: $Q_1 \times \{C,U\} \times Q_m$ to keep a state of M_m during both passes,
- alphabets of tapes: $Q_1 \times \{C,U\} \times Q_m \times \Gamma_1 \times \Gamma_2 \times \ldots \times \Gamma_k$ for symbols under heads during both passes,
- markers visited heads, this might be done by extending tapes' alphabets with a special *marker-not-visited-yet* symbol $\hbar$: $Q_1 \times \{C,U\} \times Q_m \times \Gamma_1' \times \Gamma_2' \times \ldots \times \Gamma_k'$, where $\Gamma_i' = \Gamma_i \cup \{\hbar\}$ for any tape i,
- a value of the transition function δ_m is stored in already included components $Q_m \times \Gamma_1' \times \Gamma_2' \times \ldots \times \Gamma_k'$ with *update* pass indication,
- heads' shifts $Q_1 \times \{C,U\} \times Q_m \times \Gamma_1' \times \Gamma_2' \times \ldots \times \Gamma_k' \times \{L,R,S\}^k$
- update heads' positions counters would require four symbols $Q_1 \times \{C,U\} \times Q_m \times \Gamma_1' \times \ldots \times \Gamma_k' \times \{L,R,S\}^k \times \{X^i, X^{ii}, X^{iii}, X^{iv}\}^k$ for every tape,
- and more states is required to organize simulation details.

Thus, states could be labelled by elements of the above Cartesian product, i.e. by (3k+3)-tuples. So then, the number of states is not less than
$r = 2 * 3^k * 4^k * |Q_1| * |Q_m| * |\Gamma_1| * |\Gamma_2| * \ldots * |\Gamma_k|$

On the other hand, symbols of the tape alphabet of a simulating machine could be denoted by all tuples of the Cartesian product of tapes' alphabets and symbols stored in tracks with heads' markers (the blank symbol B and the head's marker symbol H) $\{H,B\} \times \Gamma_1 \times \{H,B\} \times \Gamma_2 \times \ldots \times \{H,B\} \times \Gamma_k$. Thus, the number of tape symbols of simulating machine is equal to $c = 2^k * |\Gamma_1| * |\Gamma_2| * \ldots * |\Gamma_k|$.

Finally, size of the transition table of the simulating machine is not less than $r * c$ comparing to size of the transition table of a multi-tape Turing machine, which is equal to $|Q_m| * (|\Gamma_1| * |\Gamma_2| * \ldots * |\Gamma_k|)$.

Let us analyze possible increase of number of transitions. The worse case is when the head of one tape of a multi-tape Turing machine always shifts right and the head

of another tape always shifts left. Tape slices with these two heads' markers will be separated by:

- one tape slices after simulation of the first transition,
- three slices after simulation of two transitions,
- $2*n-1$ tape slices after simulation of n transitions.

Simulation of transitions of a multi-tape Turing machine, as described above, requires following numbers of transitions of simulating machine:

- three transitions for simulation of the first transition,
- at least seven transitions for simulation of the second transition,
- at least $4*n-1$ transitions for simulation of the $n-th$ transition. This number will be increased by transitions updating markers of those heads of a multi-tape machine, which shift right. The increment will not exceed $2*(k-2)$, where k is the number of tapes.

As a result, simulation of n transitions of a multi-tape Turing machine requires not less than

$$f(n) = 3+7+11+\ldots+(4*n-1)+n*(k-2) =$$

$$4+8+\ldots+4*n+(k-3)*n \;=\; 2*n^2+(k-1)*n$$

transitions of simulating machine. This number could be estimated:
$2*n^2 \leq f(n) \leq 3*n^2$ for n big enough. Therefore, we can say that simulation increases cost of computation with square.

Definition of a configuration (step description) of a multi-tape machine must consider contents of several tapes and one state of control unit:

Definition 4.16. A configuration (step description) of a multi-tape Turing machine with k-tapes is the following sequence of symbols

$$(\alpha_1^1, \alpha_1^2, \ldots, \alpha_1^k)\; q\; (\alpha_2^1, \alpha_2^2, \ldots, \alpha_2^k)$$

where:

- q is a state of control unit,
- α_1^i is a sequence of symbols for i-th tape in cells preceding the head (like in Turing machines with two-way infinite tape),
- α_2^i is a sequence of symbols for i-th tape in cells from the head's right (like in Turing machines with two-way infinite tape).

In practice, step description will be shown in two ways presented below at examples:

- state of the control unit plays a role of tape markers, contents of tapes is split for left and right sequences of symbols and contents of tapes may be shifted each other:

$$\begin{array}{rcl} X_1^1X_2^1X_3^1 & & X_4^1X_5^1X_6^1X_7^1 \\ X_1^2X_2^2X_3^2X_4^2X_5^2 & q & X_6^2X_7^2 \\ & & X_1^3X_2^3X_3^3X_4^3 \end{array}$$

- state of the control unit is placed at the beginning, contents of tapes are fixed together, heads' positions are underscored:

$$\begin{array}{rl} & X_1^1X_2^1X_3^1X_4^1X_5^1X_6^1X_7^1 \\ q & X_1^2X_2^2X_3^2\underline{X_4^2}X_5^2\underline{X_6^2}X_7^2 \\ & X_1^3X_2^3X_3^3X_4^3 \end{array}$$

4.2 Nondeterministic Turing machines

Nondeterministic model is a very important modification of Turing machines. Equivalence of deterministic and nondeterministic Turing machines is the most important conclusion drawn from discussion in this Chapter.

As mentioned before, for any configuration of deterministic Turing machines, at most one transition could be done. Unlike, nondeterministic Turing machines allow for a choice between several transitions for some (or all) configurations. This means that transition function may yield a set of possible transitions. In this section we provide a definition of nondeterministic Turing machine in basic model. Then we prove equivalence between nondeterministic Turing machines in basic model and a multi-tape deterministic Turing machines. A study on equivalence of different models of nondeterministic Turing machines is similar to analogous study on deterministic Turing machines. For that reason we skip over such a discussion.

Definition 4.17. Nondeterministic Turing machine in basic model is a system

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F, C)$$

with components as follows:

- δ is the transition function: $\delta : Q \times \Gamma \to \bigcup (Q \times \Gamma \times \{L,R\})^k$
  where $(Q \times \Gamma \times \{L,R\})^0$ stands for an undefined value of transition function and $(Q \times \Gamma \times \{L,R\})^k$ denotes the set of values of transition function consisting of k 3-tuples being transition descriptions $(p,X,D) \in Q \times \Gamma \times \{L,R\})$.
- descriptions other components are given in Definition 4.1.

Note that values of transition function are sets of several descriptions of transitions rather than one description (like in case of deterministic Turing machines), i.e. we may have a set of k-elements as a value of the transition function:

$$\delta(q,X) = \{(p_1,Y_1,D_1),(p_2,Y_q,D_2),\ldots,(p_k,Y_k,D_k)\}$$

We will informally use a term that *transition function yields k values*.

Once the empty set or a one element set $\{(p,Y,D)\}$ is a value of the transition function, then transition is performed like for deterministic machine, i.e.:

1. if transition function yields the empty set, i.e. it is undefined, then machine falls into infinite computation,
2. for a value $\{(p,Y,D)\}$ of transition function, the control unit switches to state p, the head stores symbol Y and shifts in direction D.

However, when transition function yields a set of $k > 1$ values then a transition could be explicated as follows:

3 a choice is made between possible k values,
4 for the chosen description a transition is made like for deterministic Turing machine,
5 it is assumed that transition chosen in the first point leads to such a computation (in the sense of deterministic Turing machines), which terminates in an accepting state, if such a computation exists.

A question can be raised, how to hold the assumption of point 5, when a choice is done. Nondeterminism does not answer this question, it just assumes such choices.

Remark 4.4. The assumption made in point 5 of the above description is the fundamental assumption of nondeterminism. It creates an interpretation of nondeterminism assuming that computation is a sequence of configurations achieved with *correct* choices between possible transitions.

Remark 4.5. Another interpretation of nondeterminism is quite practical. When transition function yields a set of $k > 1$ values, the machine creates k copies of itself. Then, every copy makes a transition matching to one value and continues its own computation. The machine accepts if and only if there is such a copy created during computation, which terminates its own computation in an accepting state.

In spirit of the last interpretation, a nondeterministic Turing machine would be interpreted as a group of Turing machines. This group includes the original machine at the start of computation and might be enlarged during computation. Every machine of this group creates copies of itself, as many copies as the number of values yielded by the transition function. Then every copy makes a transition, as described above. Note, that every copy is doing computation like a deterministic Turing machine.

Let us notice, that step description of a nondeterministic Turing machine is exactly the same like for a deterministic one (of the same model). However, notions of transition relation and computation must be re-defined for nondeterministic Turing machines.

Definition 4.18. A pair of configurations of a nondeterministic Turing machine is in the transition relation if and only if the second configuration can be derived from the first one by application of a transition yielded by the transition function, c.f. Definition 4.3.

Definition 4.19. A computation of a Turing machine $M = (Q, \Gamma, \Sigma, \delta, q_0, B, F, C)$ is a tree such that:

- its nodes are labelled by configurations of the machine,
- its root is labelled by the initial configuration,
- for any node η_p, its every child η_c is related to it in the transition , relation, i.e. $\eta_p \succ \eta_c$.

Note that a computation tree is a k-tree, where k is the maximal number of values yielded by the transition function for given arguments. We say that degree of nondeterminism is k.

Computation of a nondeterministic Turing machine is a tree, which may have finite paths form the root to leaves as well as infinite paths beginning in the root. Interpreting a nondeterministic Turing machine as a group of its copies, we can associate finite paths with corresponding copies of the machine. Based on this interpretation we can say that a nondeterministic Turing machine accepts its input if and only if there is a copy of the machine, which terminates its computation in an acting state.

Definition 4.20. A nondeterministic Turing machine accepts its input if and only if the computation tree has a path from the root to a leaf corresponding to such a configuration, for which the control unit is in an accepting state.

Proposition 4.8. *Nondeterministic Turing machines are equivalent to deterministic Turing machines.*

Proof. We will show that for a given Turing machine of one type, an equivalent machine of another type can be designed.

Note that a deterministic Turing machine in basic model is also a nondeterministic one (with the maximal number of values yielded by transition function not greater than 1). For that reason and due to equivalence of different types of Turing machines, any deterministic Turing machine is equivalent to some nondeterministic one.

Now, for a given nondeterministic Turing machine in basic model an equivalent deterministic multi-type Turing machine will be built.

Let us present an idea of construction of a deterministic Turing machine equivalent to a nondeterministic one. The idea is based on a deterministic simulation of a computation of a nondeterministic Turing machine. It can be briefly presented as follows:

- a nondeterministic Turing machine accepts its input if and only if the computation three has a node, which terminates computation in an accepting state,
- a computation tree is a k-tree, where k is the maximal number of values yielded by the transition function,
- a breadth-first tree searching algorithm, which starts searching from the root and, then, visits nodes level by level, will eventually visit a node, which terminates computation in an accepting state,

- a simulation of a computation of a nondeterministic machine is rooted in the breadth-first searching. This simulation investigates nodes visited by breadth-first search. For a visited node the computation from the root to this node is reproduced. If this computation terminates in an accepting state, then simulation is terminated with acceptation. Otherwise, breadth-first searching is continued and a next node is investigated.

Note that the above method replicates transitions many times: closer a node to the root, more replications is done for transitions of the path from the root to this node. This method has huge complexity of computation. However, a more efficient deterministic method to simulate nondeterminism is not known.

The above idea can be realized by a 3-tape deterministic Turing machine. Let us assume that degree of nondeterminism of a simulated nondeterministic Turing machine is r. The simulation algorithm could is briefly described as follows:

- an input of a nondeterministic machine is stored on the first tape,
- the simulating deterministic machine systematically generates all possible sequence of length being successive natural numbers $0,1,2,\ldots$. Natural numbers from the interval $[1,2,\ldots,r]$ are elements of generated sequences. Every generated sequence is stored on the second tape,
- for every generated sequence the following computation is done:
  - the input is copied from the first tape to the third tape,
  - a path from the computation of nondeterministic Turing machine is simulated. The path begins in the root of the computation tree and is defined by the sequence $(i_1,i_2,\ldots,i_n)$:
    - length of the path is equal to length of the sequence, i.e. n transitions are simulated,
    - the successive $l-th$ transition is defined by the $l-th$ value of the transition function of the nondeterministic Turing machine. If the transition function yields less than k values for given arguments and the $l-th$ value has no corresponding transition, than simulation is broken and the algorithm goes to generating and investigating a next sequence of numbers,
  - if, for given generated sequence, a node terminating computation of the nondeterministic Turing machine is reached, then simulating machine stops its computation and accepts the input. Otherwise, the algorithm goes to generating and investigating a next sequence of numbers.

The above simulation algorithm describes a deterministic Turing machine, which is equivalent to the simulated nondeterministic Turing machine.

4.3 Linear bounded automata

We can observe that many Turing machines can exploit during computation only this part of its tape, which was used to store input data. Indeed, such Turing machines are

distinguished as a subclass called linear bounded automata. In the book, only linear bounded automata, which have the stop property or are equivalent to them, will be considered. Such automata will be used to accept context-sensitive languages. The formal definition of linear bounded automata is as follows:

Definition 4.21. A linear bounded automaton in basic model is a Turing machine (nondeterministic, in general) with the stop property:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, \#, \&, F, C)$$

with components as follows:

- $\#, \&$ are the left and the right guard, $\#, \& \in \Gamma$, $\#, \& \notin \Sigma$,
- $\#q_0 a_1 a_2 \ldots a_n \&$ is the initial configuration, where $a_1 a_2 \ldots a_n$ is input data,
- the transition function cannot yield a value that allows the head:
  - to replace the left guard symbol with any other symbol or to make a shift left when it reads the left guard symbol, i.e.
    $\delta(q, \#) = \{(p_1, \#, R), (p_2, \#, R), \ldots, (p_k, \#, R)\}$ for any $q, p_1, p_2, \ldots, p_k \in Q$,
  - to replace the right guard symbol with any other symbol or to make a shift right when it reads the right guard symbol, i.e.
    $\delta(q, \&) = \{(p_1, \&, L), (p_2, \&, L), \ldots, (p_k, \&, L)\}$ for any $q, p_1, p_2, \ldots, p_k \in Q$,
  - to store a guard symbol in any cell besides these holding them, i.e.
    if $X \in \Gamma - \{\#, \&\}$ and $q \in Q$,
    then $\delta(q, X) = \{(p_1, Y_1, D_1), (p_2, Y_2, D_2), \ldots, (p_k, Y_k, D_k)\}$,
    where $Y_1, Y_2, \ldots, Y_k \in \Gamma - \{\#, \&\}$, $p_1, p_2, \ldots, p_k \in Q$ and
    $D_1, D_2, \ldots, D_k \in \{L, R\}$,
- other components are as in Definition 4.1.

The discussion on varieties of Turing machines could be adapted to analysis of linear bounded automata. However, not all types of Turing machines have counterparts of linear bounded automata. For instance, Turing machines in basic model or with two-way infinite tape do not have equivalent linear bounded automata. On the other hand, the idea of guard furnishing Turing machines is common with the definition of linear bounded automata, though the later ones have more restrictions than former ones. Such ideas utilized for Turing machines as: halting states and a multi-track tape may be directly adjusted to linear bounded automata. Also, the idea of multi-tape is adaptable to linear bounded automata in sense that all tapes have length equal to input data and have the left and the right guard.

There is a model, which raised the name of linear bounded automata. In this model length of the tape is bounded by a linear function of the length of the input data or, equivalently, is a multiply of the length of the input data. In these automata, the beginning and the end of tape is marked by guards, as in basic model.

Proposition 4.9. *The following classes of deterministic linear bounded automata are equivalent:*

- *in basic model,*
- *with halting accepting state,*
- *with the length of the tape bounded by a linear function,*
- *with a multi-track tape,*
- *a multi-tape*

The following classes of nondeterministic linear bounded automata are equivalent:

- *in basic model,*
- *with halting state,*
- *with the length of the tape bounded by a linear function,*
- *with a multi-track tape,*
- *a multi-tape.*

The same classes of nondeterministic linear bounded automata are equivalent.

Proof. A proof of equivalence of these classes is quite similar to proofs of equivalence of respective classes of Turing machines.

Chapter 5
Pushdown automata

Pushdown automata varies form Turing machines in their definition and interpretation. However, despite differences, we will prove in this Chapter, that pushdown automata are limited Turing machines. Pushdown automata are finite structures with a stack, which is a potentially infinite element. A stack is a data structure, also called LIFO, i.e. "last in, first out", which allows for storing abstract elements of data and removing them. Usually two operations are used for operating a stack: *push* and *pop*. The *push* operation adds an element to the stack hiding elements previously or initializes the stack, if it is empty. The *pop* operation removes and returns the element most recently added to the stack or returns the empty value if the stack is empty (this is why stack is also called LIFO structure). Note that there elements besides of the stack are not accessible except the one on the top of the stack. In order to get access to a requested element formerly pushed on the stack, it is necessary to pop all elements pushed later than the requested one. In other words, elements are removed from the stack in the reverse order to the order of their addition. Thus, stack data accessibility is significantly limited comparing to tapes of Turing machines. As a result, pushdown automata are less powerful than Turing machines.

5.1 Nondeterministic pushdown automata

We discuss nondeterministic pushdown automata first. Deterministic ones should fulfill conditions clearer on the basis of general definition of nondeterministic ones.

Definition 5.1. A pushdown automaton is a system

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \triangleright, F, R)$$

with components as follows:

Q - a finite set of states,
Γ - a finite set of stack symbols (stack alphabet),

$\triangleright$ - an initial stack symbol, $\triangleright \in \Gamma$,
Σ - a finite input alphabet,
q_0 - the initial state, $q_0 \in Q$,
F - a set of accepting states, $F \subset Q$,
R - a set of rejecting states, $R \subset Q$, $F \bigcap R = \varnothing$,
δ - a transition function, $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \bigcup_{k=0}^{\infty}(Q \times \Gamma^*)^k$

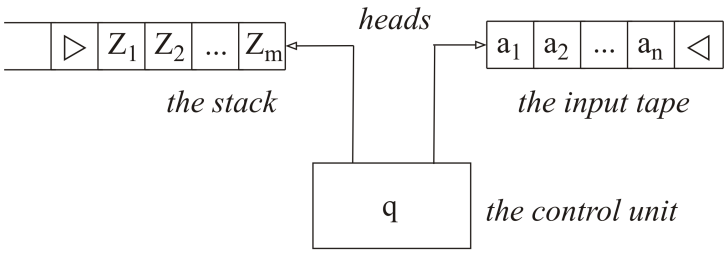

Fig. 5.1 Pushdown automaton.

A pushdown automaton could be interpreted as a physical mechanism shown in Figure 5.1.This mechanism consists of:

- a control unit, it is in a state of $q \in Q$,
- an input tape holding input data $a_1 a_2 \ldots a_n$, the special end-of-input symbol $\triangleleft$ is attached to input data, the symbol $\triangleleft$ neither belongs to the input alphabet, nor to the stack alphabet, it marks the end of input easing practicing pushdown automata,
- an input head, which reads an input symbol and shifts right or does not take any action,
- a stack, which is a one-way infinite tape with stack structure, which gives access only to its top cell, i.e. its first cell,
- a stack head, which can perform *push* and *pop* operations.

A pushdown automaton is aimed on accepting a language, i.e. on answering if its input word is in the language or not. Computation of a given pushdown automaton is done according to the following intuitive procedure:

1. the initial configuration of a given pushdown automaton is described as follows:
   a. an input data, a word $w = a_1 a_2 \ldots a_n$ over input alphabet Σ, is stored on the input tape, c.f. Figure 5.1,
   b. an end-of-input symbol $\triangleleft$ is attached to the input data,
   c. the head of the input tape is placed over the first (leftmost) symbol of the input word,
   d. the head of the stack is (always) placed over the top cell of the stack
   e. the control unit is in the initial state q_0,

2. if the input head reads the end-of-input symbol $\lhd$ and the control unit is in a accepting or rejecting state, then computation is terminated and the automaton responds its state, i.e. if a state of its control unit is an accepting one or a rejecting one,
3. if conditions of point 2 are not satisfied, then based on

   a. a state q of the control unit,
   b. a symbol X read by the head of the stack,
   c. either an input symbol a or without it,

   the automaton makes the following actions:

   d. values $\{(p_1,\alpha_1),(p_2,\alpha_2),\ldots,(p_k,\alpha_k)\}$ of the transition function $\delta(q,a,X)$ or $\delta(p,\varepsilon,X)$ are computed, where $a \in \Sigma$ is the input symbol, $X \in \Gamma$ is the top symbol of the stack, $q,p_1,p_2,\ldots,p_k \in Q$ are states, $\alpha_1,\alpha_2,\ldots,\alpha_k \in \Gamma^*$ are strings of symbols of the stack alphabet,
   e. if values of the transition function are computed based on an input symbol $\delta(q,a,X)$, then the input head is shifted right, otherwise the input head does not change its position,
   f. a value (p_i,α_i) of the transition function is chosen nondeterministically,
   g. the top symbol X of the stack is removed and then symbols of the string α_i are pushed on the stack in reverse order, i.e. the last symbol first, the first one last. The first symbol of α_i will be on the top of the stack after this operation,
   h. the control unit switches to the state p_i,

4. computation goes to the point 2.

We assume that pushdown automata always terminate computation. The value $(Q \times \Gamma^*)^0$ of a transition function, which is the empty word ε, is interpreted as termination of computation in a rejecting state.

Note that pushdown automata cannot store output information, so - unlike Turing machines - they can only accept languages and cannot compute functions or solve problems.

Now we define step a description, a transition relation and a computation of pushdown automata.

Definition 5.2. A step description (a configuration) of a pushdown automaton $M = (Q,\Sigma,\Gamma,\delta,q_0,\rhd,F,R)$ is the following sequence of symbols:

$$\gamma\, q\, w$$

where:

- $q \in Q$ is the current state of the control unit of a pushdown automaton,
- γ is the stack contents, the last symbol of γ is the top symbol of the stack,
- ω is the current input, the first symbol of w is the symbol under the input head.

Note that both γ and ω sequences of symbols are words over the stack alphabet Γ and the input alphabet Σ and that any of these sequences may include only the

initial stack symbol $\triangleright$ and the end-of-input symbol $\triangleleft$. However, none of these two sequences can be infinite, like in a case of a step description of a Turing machine.

For instance, the initial step description (configuration) is of the form $\triangleright\ q_0\ w\ \triangleleft$, where q_0 is the initial state, w is the input data. In this case the stack is empty, so the initial stack symbol is near the initial state symbol. On the other hand, a step description $\gamma\ q\ \triangleleft$ informs that input symbols have already been read.

Let us analyze transitions done by pushdown automata. We assume that a step description is described by the following sequence of symbols:

$$\triangleright\ X_1\ X_2\ \ldots\ X_{m-1}\ X_m\ \ q\ \ a_i\ a_{i+1}\ \ldots\ a_n\ \triangleleft$$

Recall that:

- q is the current state of the control unit,
- $\triangleright\ X_1\ X_2\ \ldots\ X_{m-1}\ X_m$ is the sequence of symbols on a stack, X_m is the top symbol of a stack,
- $a_i\ a_{i+1}\ \ldots\ a_n\ \triangleleft$ is the sequence of input symbols, a_i is the front input symbol
- the head of a stack reads X_m,
- the input head either reads a_i, or does not read anything.

The next step description is determined by the transition function:

- if the value of the transition function is $\delta(q, a_i, X_m) = \{(p_1, \varepsilon), (p_2, Y_2^1\ Y_2^2\ Y_2^3)\}$, for instance, and the second value $(p_2, Y_2^1\ Y_2^2\ Y_2^3)\}$ is chosen nondeterministically, then the following step description is yielded,

$$\triangleright\ X_1\ X_2\ \ldots\ X_{m-1}\ Y_2^3\ Y_2^2\ Y_2^1\ \ p_2\ a_{i+1}\ \ldots\ a_n\ \triangleleft$$

- if the value of the transition function is $\delta(q, \varepsilon,\ X_m) = \{(p_1, \varepsilon), (p_2, Y_2^1\ Y_2^2\ Y_2^3)\}$, for instance, and the first value (p_1, ε) is chosen nondeterministically, then the following step description is yielded:

$$\triangleright\ X_1\ X_2\ \ldots\ X_{m-1}\ p_1\ a_i\ a_{i+1}\ \ldots\ a_n\ \triangleleft$$

We will use the symbol $\succ$ to denote a transition of a pushdown automaton. The transition symbol $\succ$ may be supplement with a pushdown automaton name $\succ_M$ to emphasize that a transition concerns a given pushdown automaton. It also can be supplemented with a superscript $\succ^k$ to notify k transitions done.

The above two transitions done by a pushdown automaton will be denoted as follows:

$$\triangleright\ X_1\ \ldots\ X_{m-1}\ X_m\ q\ a_i\ a_{i+1}\ \ldots\ a_n\ \triangleleft\ \ \succ\ \ \triangleright\ X_1\ \ldots\ X_{m-1}\ Y_2^3\ Y_2^2\ Y_2^1\ \ p_2\ a_{i+1}\ \ldots\ a_n\ \triangleleft$$

$$\triangleright\ X_1\ \ldots\ X_{m-1}\ X_m\ q\ a_i\ a_{i+1}\ \ldots\ a_n\ \triangleleft\ \ \succ\ \ \triangleright\ X_1\ \ldots\ X_{m-1}\ p_1\ a_i\ a_{i+1}\ \ldots\ a_n\ \triangleleft$$

Definition 5.3. Transitions of a pushdown automaton create a binary relation in the space of all possible configurations of the automaton, i.e. any two configurations are related if and only if the second is derived from the first one by application of a

transition function. This relation is called the transition relation of a given pushdown automaton. The transitive closure of the transition relation is denoted $\succ^*$.

Definition 5.4. A computation of a pushdown automaton is a tree such that:

- its nodes are labelled by configurations of an automaton,
- its root is labelled by the initial configuration,
- for any node η_p, its every child η_c is related to it in the transition, relation, i.e. $\eta_p \succ \eta_c$.

Note that a computation tree is a (k+1)-tree, where k is the maximal number of values yielded by the transition function for given arguments. Note that a transition for a given state and a given stack symbol can be chosen form k-transitions respective to an input symbol and one transition done without checking an input.

Now we give a formal definition of acceptation of an input by a pushdown automaton.

Definition 5.5. A pushdown automaton accepts its input if and only if the pair of the initial configuration and a final configuration belongs to transitive closure of a transitive relation i.e. $\eta_1 \succ^* \eta_T$, where η_1 is an initial configuration and η_T is a final configuration.

Remark 5.1. A pushdown automaton accepts its input if and only if the computation tree has a path from the root to a leaf labelled by an accepting configuration.

Based on the above discussion we give now formal definitions of some concepts.

Definition 5.6. The language accepted by a pushdown automaton is the set of words $w \in \Sigma^*$ accepted by a pushdown automaton.

5.2 Deterministic pushdown automata

A pushdown automaton is a deterministic one if there is no more than one possible transition in any configuration. This condition needs that, for any configuration of an automata, a transition function yields at most one possible transition. However, for push down automata, there is another nondeterministic factor: for a given state and a given stack symbol there might be a choice between transitions for given input symbols and without involving an input symbol (ε-symbol). As a result, the following definition formulates conditions for a pushdown automata to be a deterministic one.

Definition 5.7. A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, \rhd, F, R)$ is a deterministic one if and only if:

- its deterministic function for any arguments, i.e. for any triple $(q,X,X),\ q \in Q,\ X \in (\Sigma \cup \{\varepsilon, \lhd\}),\ X \in \Gamma$, yields at most one transition,

- for given $q \in Q$ and $X \in \Gamma$ the transition function rejects for (q, ε, X) or rejects for all (q, a, X), $a \in (\Sigma \cup \{\varepsilon\})$.

Proposition 5.1. *Deterministic pushdown automata are not equivalent to nondeterministic ones.*

5.3 Accepting states versus empty stack

Examples presented in previous sections show that acceptation came with the initial symbol of the stack $\triangleright$ and the end-of-input symbol $\triangleleft$. A question may be asked, if this is a coincidence or rather a rule, that we can change acceptation by accepting state to acceptation by empty stack (empty input is default, because we already assumed that acceptation must be accompanied with empty input). This question can be answered positively. We may move up a class of pushdown automata, which accept when the stack is empty, and prove that automata accepting by empty stack are equivalent to automata accepting with a state. An idea of a proof is quite clear. On one hand, having a pushdown automaton accepting with a state, we can empty its stack when an accepting state is reached and then accept its input. On the other hand, given a pushdown automaton accepting with an empty stack, it should make additional transition to an extra accepting state, when its stack is empty. Details are given below.

Definition 5.8. A pushdown automaton accepting by empty stack is a system

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \triangleright, \varnothing, R)$$

where the set of accepting states is empty and other components are as shown in Definition 1.

Note that acceptance by empty stack does not depend on a state of a terminated automaton description. Because of this the set of accepting states of such an automaton is empty.

Proposition 5.2. *Acceptance by states is equivalent to acceptance by empty stack.*

Proof. We prove that for any pushdown automaton accepting by states there exists an equivalent automaton accepting by empty stack and oppositely.

Assume that there is a pushdown automaton accepting by states

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \triangleright, F, R)$$

The following pushdown automaton accepting with empty stack is equivalent to the above one

$$M' = (Q', \Sigma, \Gamma', \delta', q_0', \triangleright', \varnothing, R)$$

where:

- $Q' = Q \cup \{q'_0, q'\}$ and $Q \cap \{q'_0, q'\} = \varnothing$,
- $\Gamma' = \Gamma \cup \{\triangleright'\}$ and $\Gamma \cap \{\triangleright'\} = \varnothing$,
- the transition function δ' is design as follows:
  - $\delta'(q'_0, \varepsilon, \triangleright') = \{(q_0, \triangleright\triangleright')\}$,
  - $\delta'(q, a, X) = \delta(q, a, X)$ for $q \in Q$, $a \in \Sigma$, $X \in \Gamma$,
  - $\delta'(q, \varepsilon, X) = \delta(q, \varepsilon, X)$ for $q \in (Q - F)$, $X \in \Gamma$,
  - $\delta'(q, \varepsilon, X) = \{(q', \varepsilon)\}$ for $q \in F$, $X \in \Gamma - \{\triangleright\}$ start emptying the stack,
  - $\delta'(q', \varepsilon, X) = \{(q', \varepsilon)\}$ for $X \in \Gamma - \{\triangleright\}$ continue emptying the stack,
  - $\delta'(q', \triangleleft, \triangleright) = \{(q', \varepsilon)\}$, remove the end-of-input symbol and the initial stack symbol, accept with empty stack and empty input,
  - M' rejects for all other configurations.

The automaton M' goes to the initial configuration of M in its first transition, c.f. the first point. Then, it follows computation of M. When M reaches an accepting configuration (recall, that it terminates computation for an accepting state), then M' pops a top symbol from its stack and switches to extra state q' to empty the stack.

Now, we design a pushdown automaton accepting by states equivalent to a given one accepting by empty stack. Assume that there is a pushdown automaton accepting empty stack

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \triangleright, \varnothing, R$$

The following pushdown automaton accepting with states is equivalent to the above one:

$$M' = (Q', \Sigma, \Gamma', \delta', q'_0, \triangleright', F, R)$$

where:

- $Q' = Q \cup \{q'_0, q_A\}$ and $Q \cap \{q'_0, q_A\} = \varnothing$,
- $\Gamma' = \Gamma \cup \{\triangleright'\}$ and $\Gamma \cap \{\triangleright'\} = \varnothing$,
- $F = \{q_A\}$,
- the transition function δ' is design as follows:
  - $\delta'(q'_0, \varepsilon, \triangleright') = \{(q_0, \triangleright\triangleright')\}$,
  - $\delta'(q, a, X) = \delta(q, a, X)$ for $q \in Q$, $a \in \Sigma$, $X \in \Gamma$,
  - $\delta'(q, \varepsilon, X) = \delta(q, \varepsilon, X)$ for $q \in Q$, $X \in (\Gamma - \{\triangleright\})$,
  - $\delta'(q, \varepsilon, \triangleright) = \{(q_A, \varepsilon)\}$ for $q \in F$,
  - M' rejects for all other configurations.

The automaton M' goes to the initial configuration of M in its first transition, c.f. the first point. Then, it follows computation of M. When M comes to its stack, then M' pops a top symbol from its stack (it is the initial symbol of M) and switches (nondeterministically) to the accepting state q_A and accepts if its input is empty. Otherwise, when input is not empty, M' continues simulation of M.

5.4 Pushdown automata as Turing machines

Pushdown automata are restricted Turing machines. Moreover, since pushdown automata always terminate their computation, they are restricted Turing machines with the stop property.

Proposition 5.3. *There is a Turing machine with the stop property equivalent to a given pushdown automaton.*

Proof. Let us design a Turing machine equivalent to a given pushdown automaton accepting with states:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \triangleright, F, R)$$

We design a 2-tape Turing machine with terminating states, which is equivalent to the automaton M:

$$M_T = (Q_T, \Sigma, \Gamma_T^1, \Gamma_T^2, \delta_T, B, \{q_A\}, \{q_R\})$$

where:

- $Q_T = (Q - (F \cup R)) \cup \{q_A, q_R\} \cup Q_S$, $Q \cap (\{q_A, q_R\} \cup Q_S) = \varnothing$ - the set of states of the Turing machine M_T includes:
  - states of the automaton M except accepting and rejecting states (except states, which terminate computation of the automaton),
  - a new halting accepting state and a new halting rejecting state of the Turing machine M_T and
  - additional states, which simulate transitions of the automaton M,
- $\Gamma_T^1 = \Sigma \cup \{\triangleleft, B\}$ - an alphabet of the first tape,
- $\Gamma_T^2 = \Gamma \cup \{\triangleright, B\}$ - an alphabet of the second tape,
- B - the blank symbol of both tapes.

A configuration of a pushdown automaton is characterized by the following configuration of a Turing machine, c.f. Figure 5.2:

- an input is stored on the first tape, the head of the first tape is placed over the first input symbol,
- a stack is stored on the second tape, the initial stack symbol is the leftmost nonblank symbol of the tape, a top stack symbol is a rightmost nonblank tape symbol, the head of the tape is placed on a rightmost nonblank symbol (a top symbol of the stack),
- the control unit of the Turing machine is in the same state as the control unit of the pushdown automaton.

All values of the transition function δ of the automaton are enumerated. Namely, if the transition function δ yields k transitions for a given triple of arguments: $\delta(q,a,X) = \{(p_1,\alpha_1),(p_2,\alpha_2),\ldots,(p_k,\alpha_k)\}$ $\delta(q,a,X)$, then every pair (p_i,α_i) gets its own number.

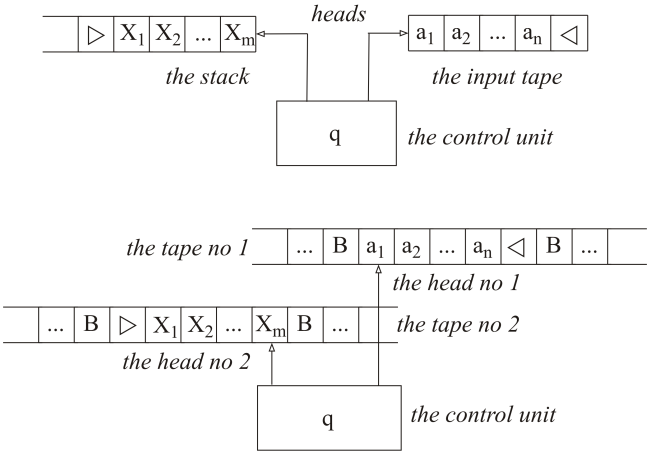

Fig. 5.2 A push down automaton and its characterization by a Turing machine.

A transition of the automaton $(p,\alpha) \in \delta(q,a,X)$ with a given number t and $p \in Q-(F \cup R)$, $a \in \Sigma \cup \{\lhd\}$ and $X \in \Gamma$ is simulated by the Turing machine as follows:

- if α is the empty sequence, then $\left(p, \begin{matrix} B & R \\ B & L \end{matrix}\right) \in \delta'\left(q, \begin{matrix} a \\ X \end{matrix}\right)$,
- if $\alpha = X_1X_2\ldots X_r$, then the set of states Q_T is expanded by adding states $p_1^t, p_2^t, \ldots, p_r^t$ and the following transitions are included:
  - $\left(p_r^t, \begin{matrix} a & S \\ X_r & R \end{matrix}\right) \in \delta'\left(q, \begin{matrix} a \\ X \end{matrix}\right)$,
  - $\delta'\left(p_r^t, \begin{matrix} a \\ B \end{matrix}\right) = \left\{\left(p_{r-1}^t, \begin{matrix} a & S \\ X_{r-1} & R \end{matrix}\right)\right\}$,
  - $\delta'\left(p_{r-1}^t, \begin{matrix} a \\ B \end{matrix}\right) = \left\{\left(p_{r-2}^t, \begin{matrix} a & S \\ X_{r-2} & R \end{matrix}\right)\right\}$,
  - $\ldots$
  - $\delta'\left(p_2^t, \begin{matrix} a \\ B \end{matrix}\right) = \left\{\left(p_1^t, \begin{matrix} a & S \\ X_1 & R \end{matrix}\right)\right\}$,
  - $\delta'\left(p_1^t, \begin{matrix} a \\ B \end{matrix}\right) = \left\{\left(p, \begin{matrix} B & R \\ B & L \end{matrix}\right)\right\}$

A simulation of a transition $(p,\alpha) \in \delta(q,\varepsilon,X)$ is similar to simulation of a transition $(p,\alpha) \in \delta(q,a,X)$ with the following changes:

- if α is the empty sequence, then $\left(p, \begin{matrix} a & S \\ B & L \end{matrix}\right) \in \delta'\left(q, \begin{matrix} a \\ X \end{matrix}\right)$,
- if $\alpha = X_1X_2\ldots X_r$, then $\in \delta'\left(p_1^t, \begin{matrix} a \\ B \end{matrix}\right) = \left\{\left(p, \begin{matrix} a & S \\ B & L \end{matrix}\right)\right\}$.

In transitions shown above, if $p \in F$, then it should be replaced with q_A and if $p \in R$, then it should be replaced with q_R.

The above construction of a Turing machine simulating a pushdown automaton shows that the Turing machine has the stop property and it accepts an input if and only if the pushdown automaton accepts the same input.

Chapter 6
Finite automata

Finite automata are the simplest model of computation. They can be derived from pushdown automata by removing stack. Finite automata are finite structures without any potentially infinite element as, for instance, a stack in pushdown automata or a tape in Turing machines. Despite of these limitations, finite automata are important theoretical and practical tools. Three classes of finite automata are distinguished: deterministic, nondeterministic and those with ε-transitions.

6.1 Deterministic finite automata

Deterministic model of finite automata is the simplest one among three types: deterministic, nondeterministic and with ε-transitions. It is the simplest one in terms of a description of automata as well as of computation realized for a given input word. From now on the term *finite automata* will denote *deterministic* finite automata. Any reference to nondeterministic finite automata or finite automata *with ε-transitions* will be explicitly acknowledged.

Definition 6.1. A deterministic finite automaton is a system

$$M = (Q, \Sigma, \delta, q_0, F)$$

with the following components:

Q - a finite set of states,
Σ - a finite input alphabet,
q_0 - the initial state, $q_0 \in Q$,
F - a set of accepting states, $F \subset Q$,
δ - a transition function, $\delta : Q \times \Sigma \to Q$.

A transitions function of a finite automaton is a total function, i.e. it is defined for all its pairs of arguments (a transition function of Turing machines and pushdown automata could be undefined for some arguments).

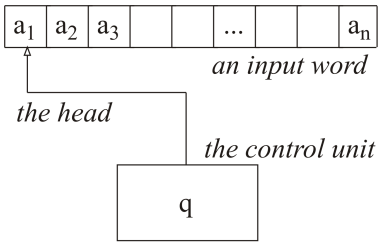

Fig. 6.1 A deterministic finite automaton.

A finite automaton could be interpreted as a physical mechanism shown in Figure 6.1. This mechanism consists of:

- a control unit, it is in a state of $q \in Q$,
- an input tape holding input data $a_1 a_2 \ldots a_n$,
- an input head, which reads an input symbol and shifts right.

Like in case of pushdown automata, a finite automaton is aimed at accepting a language, i.e., answering if its input word belongs to the language or does not. Computation of a given finite automaton is done according to the following intuitive procedure:

1. the initial configuration of a given finite automaton is described as follows:
   a. an input data, a word $w = a_1 a_2 \ldots a_n$ over input alphabet Σ, is stored on the input tape, c.f. Figure 6.1,
   b. the head of the input tape is placed over the first (leftmost) symbol of the input word,
   c. the control unit is in the initial state q_0,
2. based on:
   a. a state q of the control unit,
   b. an input symbol a

   the automaton realizes the following actions:
   c. a state p of the transition function $\delta(q,a)$ is computed, where $a \in \Sigma$ is the input symbol and $q, p \in Q$ are states,
   d. the input head is shifted right,
   e. the control unit switches to the state p,
3. if the input is not empty, then computation goes to the point 2, otherwise, computation is terminated and an automaton responds a state of the control unit.

Of course, computation of a finite automaton always terminates.

Note that finite automata, like pushdown automata, cannot store output information, so they can only accept languages and cannot compute functions or solve problems.

Now we define a step description, a transition relation and a computation of finite automata.

Definition 6.2. A step description (a configuration) of a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is the following sequence of symbols:

$$q\ \omega$$

where:

- $q \in Q$ is the current state of the control unit of a finite automaton,
- ω is the current input, the first symbol of ω is the symbol under the input head.

Note that a sequence of symbols ω is a word over the input alphabet Σ. It is empty after termination of a computation. It never can be infinite.

For instance, the initial step description (configuration) is $q_0 a_1 a_2 \ldots a_n$, where q_0 is the initial state, $w = a_1 a_2 \ldots a_n$ is the input data. On the other hand, a step description q informs that the control unit of a finite automaton is in a state q and an input symbols have already been read.

As in case of Turing machines and pushdown automata, we will use the symbol $\succ$ to denote a transition of a finite automaton. The transition symbol $\succ$ may be supplemented with a finite automaton name $\succ_M$ to emphasize that a transition concerns a given finite automaton. It also can be supplemented with a superscript $\succ^k$ to notify that k transitions done.

If a step description $q\ a_i a_{i+1} \ldots a_n$ and a transition function applicable for this step description is $\delta(q, a_i) = p$, then a next step description after a transition is made is $p\ a_{i+1} \ldots a_n$. These two step descriptions create a transition of a finite automaton, which is denoted as:

$$q\ a_i\ a_{i+1}\ \ldots\ a_n\ \succ\ p\ a_{i+1}\ \ldots\ a_n$$

Definition 6.3. Transitions of a finite automaton create a binary relation in the space of all possible configurations of the automaton, i.e. any two configurations are related if and only if the second is derived from the first one by application of a transition function. This relation is called the transition relation of a given finite automaton. The transitive closure of the transition relation is denoted by $\succ^*$

Definition 6.4. A computation of a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is a sequence of configurations $\eta_1, \eta_2, \ldots, \eta_n$ such that η_1 is the initial configuration, η_T is the final configuration and a pair of any two successive configurations belongs to the transition relation. A computation is denoted as $\eta_1\ \succ\ \eta_2\ \succ \ldots \succ\ \eta_n$.

Remark 6.1. A computation of a finite automaton is a finite sequence of configurations

$$q_0 a_1 a_2 a_3 \ldots a_n \succ q_{i_1} a_2 a_3 \ldots a_n \succ q_{i_2} a_3 \ldots a_n \succ q_{i_{n-1}} a_n \succ q_{i_n}$$

Because any transition consists of reading an input symbol and switching to a state, a computation will be shown in a simpler form

$$q_0\ a_1\ q_{i_1}\ a_2\ q_{i_2}\ a_3\ \ldots\ a_n\ q_{i_{n-1}}\ a_n\ q_{i_n}$$

Now we give a formal definition of acceptance of an input by a finite automaton.

Definition 6.5. A finite automaton accepts its input if and only if the pair of the initial configuration and a final configuration belongs to transitive closure of a transitive relation i.e. $\eta_1 \succ^* \eta_T$, where η_1 is an initial configuration and η_T is a final configuration.

Based on the above discussion we give now formal definitions of some concepts.

Definition 6.6. The language accepted by a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is the set of words $w \in \Sigma^*$ accepted by a finite automaton.

A transition relation identified in Definition 6.3 yields exactly one state for a given configuration of a finite automaton. Namely, for a given state q and a given symbol of an input alphabet a there is exactly one state p related to the given state q. The state p is yielded by a transition function: $p = \delta(q,a)$. This property comes from the definition of a transition function, which is total and its value is a state: $\delta : Q \times \Sigma \to Q$. In other words, for a given state, exactly one state is related to it with regard to a given symbol of an input alphabet. The transitive closure of a transition relation is a function as well, i.e. for a given state, exactly one state is related with regard to a given sequence of symbols of an input alphabet. This property is exploited in a definition of a so called *closure of transition function*.

Definition 6.7. A closure of a transitions function δ of a given deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is the function $\hat{\delta}$ satisfying the following conditions

1. $\hat{\delta} : Q \times \Sigma^* \to Q$,
2. $(\forall q \in Q)\ \hat{\delta}(q, \varepsilon) = q$,
3. $(\forall q \in Q)(\forall a \in \Sigma)(w \in \Sigma^*)\hat{\delta}(q, wa) = \delta(\hat{\delta}(q,w), a)$.

The closure of a transition function of a deterministic finite automaton extends a domain of a transition function form an input alphabet of an automaton to the set of all words over an input alphabet. The closure of a transition function applied to an initial configuration of a finite automaton immediately yields a result of computation $\hat{\delta}(q_0, w)$ of this automaton for the initial state q_0 and an input word $w = a_1 a_2 \ldots a_n$.

Remark 6.2. The restriction of the closure $\hat{\delta}$ of a transition function δ to the domain of δ is equal to δ:

$$\hat{\delta}\big|_{Q\times\Sigma} = \delta$$

For that reason both a transition function of a deterministic finite automaton and its closure are denoted by the same symbol δ unless it might result in some misinterpretation.

6.2 Nondeterministic finite automata

In the previous section we have studied deterministic finite automata. In this section we introduce and discuss nondeterministic finite automata. Nondeterministic finite automata compared to their deterministic counterparts differ in a form of the transition function. A transition function of nondeterministic finite automata allows for choosing a transition among several states. This new feature simplifies solutions of problems, though it does not increase computational abilities of finite automata. Below, we will prove that both classes of automata, i.e. deterministic finite automata and nondeterministic finite automata, are equivalent with regard to nature of the accepted languages. This means that every of these two classes of finite automata accept the same class of languages. In other words, for a given automaton of one class we can construct an equivalent automaton of another class. The proof of equivalency of both classes of automata is a constructive one, what means that it formulates a method of construction of a deterministic finite automaton, which is equivalent to a given nondeterministic one. Of course, a trivial opposite construction is also mentioned.

Definition 6.8. A nondeterministic finite automaton is a system

$$M = (Q, \Sigma, \delta, q_0, F)$$

where:

- δ - a transition function, $\delta : Q \times \Sigma \to 2^Q$,
- Q, Σ, q_0, F are the same as given in Definition 6.1.

A transition function of nondeterministic finite automata is a total function, the same as for deterministic finite automata, i.e., it is defined for all pairs of arguments. The values of a transition function are subsets of a set of states including the empty set (which is a subset of the set of all states).

An interpretation of a nondeterministic finite automaton is as it is shown in Figure 6.1.

A definition of a configuration (step description) of a nondeterministic finite automaton is identical with the definition of a configuration of deterministic finite automata, c.f. Definition 6.2.

Remark 6.3. Like in case of deterministic finite automata, nondeterministic finite automata are aimed at accepting a language. The computation of a given nondeterministic finite automaton is done according to the following intuitive procedure:

1. the initial configuration of a given finite automaton is described as follows:
   a. an input data, a word $w = a_1 a_2 \ldots a_n$ over input alphabet Σ, is stored on the input tape, c.f. Figure 6.1,
   b. the head of the input tape is placed over the first (leftmost) symbol of the input word,

c. the control unit is in the initial state q_0,

2. based on:

   a. a state q of the control unit,
   b. an input symbol a,

   the automaton realizes the following actions:

   c. a set of states $\{p_1, p_2, \ldots, p_k\}$ of the transition function $\delta(q,a)$ is computed, where $a \in \Sigma$ is the input symbol and $q, p_1, p_2, \ldots, p_k \in Q$ are states,
   d. if the set yielded by a transition function is empty then computation is terminated and an input is rejected, otherwise
   e. a state p_i is nondeterministically picked up from the set of states $\{p_1, p_2, \ldots, p_k\}$,
   f. the input head is shifted right,
   g. the control unit switches to the state p_i,

3. if the input is not empty then computation goes to the point2, otherwise computation is terminated and an automaton responds with a state of the control unit.

Of course, computation of a nondeterministic finite automaton always terminates, as we have seen for deterministic finite automata.

The symbols $\succ$, $\succ_M$ and $\succ^k$ are used in usual way, refer to our discussion on deterministic finite automata.

Definition 6.9. Transitions of a nondeterministic finite automaton form a binary relation in the space of all possible configurations of the automaton, i.e. any two configurations are related if and only if the second is derived from the first one by a transition in terms of point 2. of Remark 6.3. This relation is called the transition relation of a given nondeterministic finite automaton. The transitive closure of the transition relation is denoted by $\succ^*$.

Definition 6.10. A computation of a nondeterministic finite automaton

$$M = (Q, \Sigma, \delta, q_0, F)$$

is a tree such that:

- its nodes are labelled by configurations of an automaton,
- its root is labelled by the initial configuration,
- for any node η_p, its every child ηc is related to it in the transition, relation, i.e. $\eta_p \succ \eta_c$.

Remark 6.4. Note that a computation tree of a nondeterministic finite automaton is a k-tree, where k is the maximal number of values yielded by the transition function for given arguments. Moreover, edges of every level of such a tree are labelled with the same input symbol.

Remark 6.5. Formal definitions of acceptance of an input and of a language accepted by a nondeterministic finite automaton are identical with the respective definitions for deterministic finite automata, c.f. Definition 6.4 and Definition 6.6.

Remark 6.6. A nondeterministic finite automaton accepts its input if and only if the computation tree has a path from the root to a leaf labelled by an accepting configuration.

An algorithm realized by the automaton:

- the automaton stays in the state q_0 reading input symbols until a sequence of three successive 0s or three successive 1s arrives,
- when the beginning of a sequence of three successive 0s or 1s is nondeterministically encountered, a transition is made to the state q_{10} or q_{11}, respectively,
- next two consecutive 0s or 1s are counted by transitions to states q_{20} and q_A or to states q_{21} and q_A,
- when the state q_A has been reached, computation stays in this state for next coming input symbols,
- input words, for which the accepting state q_A is reached, are accepted and only such words.

The transition relation of nondeterministic automata yields a subset of a set of states for given configuration (unlike the transition relation of deterministic automata, which yields exactly a single state). That is, for a given state q and for a given symbol a of an input alphabet, states $p \in P \subset Q$ are related to the given state q. This property comes from the definition of a transition function, which is total and its values are subsets of the set of states: $\delta : Q \times \Sigma \to 2^Q$. The transitive closure of a transition relation relates as well subsets of the set Q for a given state and a given symbol of an input alphabet. The following definition provides details of a so called *closure of transition function* for nondeterministic automata.

Definition 6.11. A closure of a transitions function δ of a given nondeterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is the function:

1. $\hat{\delta} : Q \times \Sigma^* \to 2^Q$
2. $(\forall q \in Q)\, \hat{\delta}(q, \varepsilon) = \{q\}$
3. $(\forall q \in Q)(\forall a \in \Sigma)(w \in \Sigma^*)\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$

where the following notation: $\delta(P, a) = \bigcup_{p \in P} \delta(p, a)$ for any $P \in Q$ stands for $\delta(\hat{\delta}(q, w), a)$.

As is in case of deterministic finite automata, the closure of a transition function of a nondeterministic finite automaton extends a domain of a transition function form an input alphabet of an automaton to the set of all words over an input alphabet. The closure of a transition function applied to an initial configuration of a finite automaton immediately yields a result of computation $\hat{\delta}(q_0, w)$ of this automaton for the initial state q_0 and an input word $w = a_1 a_2 \ldots a_n$.

Remark 6.7. For nondeterministic finite automata, the restriction of the closure $\hat{\delta}$ of a transition δ to the domain $Q \times \Sigma$ is equal to the transition function δ:

$$\hat{\delta}\big|_{Q\times\Sigma} = \delta$$

For that reason both a transition function of a nondeterministic finite automaton and its closure are denoted by the same symbol δ unless it may cause misinterpretation.

Remark 6.8. An input word w is accepted by a nondeterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ if and only if $\delta(q_0, w) \cap F \neq \varnothing$.

A question is raised whether nondeterministic finite automata are equivalent to deterministic ones. The answer is positive.

On the one hand, a deterministic automaton is such a nondeterministic one, which does not exploit nondeterminism. Formally, a deterministic automaton can be turned to a nondeterministic one by replacing a value of a transition function of a deterministic finite automaton, which is a state, by the set including this state.

Conversely, we will prove that, for a given nondeterministic automaton, there exists an equivalent deterministic one, i.e. accepting the same language. An idea of construction of such a deterministic automaton is based on observation of computations of both types' automata. An attempt is made to turn a computation of a nondeterministic automaton, which is a tree of configurations, to sequence of configurations. States of every level of a computation tree are collected into a set of them. In such a way, a tree is twisted to a sequence of alternating sets of states and symbols of input alphabet. Such a sequence corresponds to a computation of a deterministic finite automaton supposing that sets of states represent potential states of a deterministic finite automaton.

The details behind these intuitive observations are outlined as follows:

Proposition 6.1. *Nondeterministic finite automata are equivalent to deterministic ones.*

Proof. First, a following deterministic automaton:

$$M = (Q, \Sigma, \delta, q_0, F)$$

is equivalent to a nondeterministic one:

$$M = (Q, \Sigma, \delta', q_0, F)$$

such that $\delta' : Q \times \Sigma \to 2^Q$, $\delta'(q,a) = \{\delta(q,a)\}$ for $q \in Q$ and $a \in \Sigma$.

Second, let us assume that now a nondeterministic finite automaton is:

$$M = (Q, \Sigma, \delta, q_0, F)$$

An equivalent deterministic automaton is denoted as follows:

$$M' = (Q', \Sigma, \delta', q'_0, F')$$

where:

- $Q' \cong 2^Q$ - states of the deterministic automata correspond to sets of the nondeterministic one. A state of Q' corresponding to a set of states $\{q_{i_1}, q_{i_2}, \ldots, q_{i_j}\}$ will be denotes $[q_{i_1}, q_{i_2}, \ldots, q_{i_j}]$ just to distinguish sets of states Q from states of Q',
- $q'_0 = [q_0]$ - the initial state of M' is the is the set including the initial set of M,
- $F' = \{[q_{i_1}, q_{i_2}, \ldots, q_{i_j}] \in Q' : \{q_{i_1}, q_{i_2}, \ldots, q_{i_j}\} \cap F \neq \varnothing\}$ - accepting states of M' are labelled by sets of states including an accepting state or states of M,
- $\delta'([q_{i_1}, q_{i_2}, \ldots, q_{i_j}], a) = [p_{i_1}, p_{i_2}, \ldots p_{i_k}] \Leftrightarrow \bigcup_{l=1}^{j} \delta'(q_{i_l}, a) = \{p_{i_1}, p_{i_2}, \ldots, p_{i_k}\}$ - the transition function of M' takes union of its values (which are sets of states of M), for states included in its argument (which corresponds to a set of states of M).

A formal proof of equivalence of M and M' is based on mathematical induction carried out with regard to the length of input word of both automata.

Let us prove that the closure of transition functions of both automata hold the equivalence for any $w \in \Sigma^*$ (note that the same symbol denotes both a transition function and its closure):

$$\delta'([q_0], w) = [q_{i_1}, q_{i_2}, \ldots, q_{i_j}] \Leftrightarrow \delta(q_0, w) = \{q_{i_1}, q_{i_2}, \ldots, q_{i_j}\} \quad (*)$$

1. for the word of length 0, i.e. for the empty word, the equivalence $(*)$: $\delta'([q_0], \varepsilon) = [q_0]$ and $\delta(q_0) = \{q_0\}$ is derived directly from definitions of the closure of transition functions,
2. let us check if the equivalence $(*)$ holds for a word $a_1 a_1 \ldots a_n a = wa$. In virtue of the inductive assumption we have this equivalence holds for any word $w = a_1 a_1 \ldots a_n \in \Sigma^*$. Let $a \in \Sigma$. Then, based on inductive assumption, we get:
   $\delta'([q_0], wa) = \delta'(\delta'([q_0], w), a) = \delta'([q_{i_1}, q_{i_2}, \ldots, q_{i_j}], a)$ and
   $\delta(q_0, wa) = \delta(\delta(q_0, w), a) = \delta(\{q_{i_1}, q_{i_2}, \ldots, q_{i_j}\}, a)$
   for some set of states $\{q_{i_1}, q_{i_2}, \ldots, q_{i_j}\} \subset Q$.
   Now, let us compute $\delta'([q_{i_1}, q_{i_2}, \ldots, q_{i_j}], a)$ based on the above definition of δ':
   $\delta'([q_{i_1}, q_{i_2}, \ldots, q_{i_j}], a) = [p_{i_1}, q_{i_2}, \ldots, p_{i_k}]$, where
   $\bigcup_{l=1}^{j} \delta'(q_{i_l}, a) = \{p_{i_1}, p_{i_2}, \ldots, p_{i_k}\}$
   On the other hand
   $\delta(\{q_{i_1}, q_{i_2}, \ldots, q_{i_j}\}, a) = \bigcup_{l=1}^{j} \delta'(q_{i_l}, a) = \{p_{i_1}, p_{i_2}, \ldots, p_{i_k}\}$
3. utilizing mathematical induction based on 1. and 2. we conclude that the equivalence $(*)$ holds.

Finally, let us notice that an input word is accepted by the nondeterministic finite automaton M if and only if it is accepted by the deterministic finite automaton M'. This property comes directly from the equivalence $(*)$, Remark 6.8 and definition of the set of accepting states F' of the automaton M'.

6.3 Finite automata with ε-moves

In previous sections of this Chapter we discussed two classes of finite automata: deterministic finite automata and nondeterministic finite automata. We proved that both classes are equivalent with regard to accepted languages, i.e., for an automaton in one class an equivalent automaton of another class can be constructed. The equivalence of both classes effectively helps in solving problems because we can choose an automaton from a class, which is more appropriate for a problem to be solved. Finite automata with epsilon-transitions, ε-transitions for short, create a third class of finite automata. ε-transitions are next tool, which may significantly help to solve problems. However, ε-transitions do not increase computational power of finite automata. In further parts of this section an equivalence of finite automata with ε-transitions with nondeterministic automata is proved. The proof is a constructive one, i.e. it develops method of construction of a nondeterministic finite automaton that is equivalent to a given finite automaton with ε-transitions. A construction of a finite automaton with ε-transitions that is equivalent to a given nondeterministic finite automaton is also shown.

Definition 6.12. A finite automaton with ε-transitions is a system

$$M = (Q, \Sigma, \delta, q_0, F)$$

where:

- δ - a transition function, $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$,
- Q, Σ, q_0, F stay the same as described in Definition 6.1.

A transition function of finite automata with ε-transitions is a total function defined for states and for symbols of the input alphabet and the empty word. The values of a transition function stay the same as for nondeterministic finite automata; i.e., they are subsets of a set of states. Thus, finite automata with ε-transitions are nondeterministic.

The difference between nondeterministic finite automata and finite automata with ε-transitions lays in an ability of the latter one to make a transition without checking an input symbol. Let us recall that we already considered a property ε-transitions of pushdown automata. When an ε-transition is made, an input symbols is not checked. This means that ε-transitions are made only for a given state of a current configuration. From another point of view, a configuration for ε-transition is only a state and with no input symbol.

A definition of a configuration (step description) of a finite automaton with ε-transitions is identical with the definition of a configuration of deterministic finite automata, c.f. Definition 6.2.

An obvious property $\varepsilon \circ \varepsilon = \varepsilon$ leads to an interesting idempotent operation on states for a given finite automaton with ε-transitions

$$M = (Q, \Sigma, \delta, q_0, F)$$

The operation is called epsilon-closure, ε-closure for short. It is denoted by $\varepsilon - Cl$ and defined as follows:

$$\varepsilon - Cl(q) = \{p \in Q : p = \delta^*(q,\varepsilon)\} \ \ for \ \ q \in Q$$

where:

$$\delta^* : Q \to 2^Q \ , \ \delta^*(q,\varepsilon) = \bigcup_{k=0}^{\infty} \delta^k(q,\varepsilon)$$

and

$$\delta^0(q,\varepsilon) = \{q\}$$
$$\delta^{k+1}(q,\varepsilon) = \delta(\delta^k(q,\varepsilon))$$

and

$$\delta(P,X) = \bigcup_{p \in P} \delta(p,X) \ \text{ for } \ P \subset Q, \ X \in (\Sigma \cup \{\varepsilon\})$$

The $\varepsilon - Cl$ is obviously idempotent, i.e.

$$\varepsilon - Cl(q) = \varepsilon - Cl(\varepsilon - Cl(q)) = \bigcup_{p \in \varepsilon - Cl(q)} \varepsilon - Cl(p)$$

Note that $\delta^*(q,\varepsilon) = \bigcup_{k=0}^{\infty} \delta^k(q,\varepsilon) \bigcup_{k=0}^{r} \delta^k(q,\varepsilon)$, where r not greater than the number of states of an automaton. In fact, when a transition diagram of an automaton is considered, then r is a longest path starting at q and having edges labelled with ε.

Remark 6.9. Like in case of deterministic finite automata, finite automata with ε-transitions are aimed at accepting a language. Computation of a given finite automaton with ε-transitions is done according to the following intuitive procedure:

1. the initial configuration of a given automaton is described as follows:
   a. an input data, a word $w = a_1\, a_2\, a_n$ over input alphabet Σ, is stored on the input tape,
   b. the head of the input tape is placed over the first (leftmost) symbol of the input word,
   c. the control unit is in the initial state q_0,
2. based on a state q of the control unit and on an input symbol $a \in \Sigma$:
   a. $\varepsilon - Cl(q)$ is computed,
   b. a state $q' \in \varepsilon - Cl(q)$ is nondeterministically picked up,
   c. a set of states $\{p_1, p_2, \ldots, p_k\} = \delta(q', a)$ is computed,
   d. if the set yielded by a transition function is empty then computation is terminated and an input is rejected, otherwise
   e. a state $p' \in \{p_1, p_2, \ldots, p_k\}$ is nondeterministically picked up,
   f. $\varepsilon - Cl(p')$ is computed,
   g. a state $p \in \varepsilon - Cl(p')$ is nondeterministically picked up,
   h. the input head is shifted right,

i. the control unit switches to the state p,

3. if the input is nonempty then computation proceeds to the point 2, otherwise computation is terminated and an automaton responds a state of the control unit.

Like for deterministic and nondeterministic finite automata, computation of a finite automaton with ε-transitions always terminates.

The symbol $\succ$ denotes a transition (of a finite automaton with ε-transitions) in terms of point 2. of Remark 6.9. The symbols $\succ_M$ and $\succ^k$ are used in a usual way, c.f. deterministic and nondeterministic finite automata.

Definition 6.13. A binary relation in the space of all possible configurations of the automaton with ε-transition is created by transitions of a finite automaton. The symbol $\succ^*$ denotes the transitive closure of the transition relation.

Definition 6.14. A computation of a finite automaton with ε-moves $M = (Q, \Sigma, \delta, q_0, F)$ is a tree such that:

- its nodes are labelled by configurations of an automaton,
- its root is labelled by the initial configuration,
- levels created by nodes are distinguished, the root creates the 0th level, children of the root create the 1st level etc.,
- nodes of an odd level are yielded by ε-closure applied to nodes of the previous level,
- nodes of an even level are yielded by transition function applied to nodes of the previous level,
- the bottom level has an odd number.

Remark 6.10. Note that like for nondeterministic finite automata, a computation tree of a finite automaton with ε-transitions is a k-tree, where k is the maximal number of values yielded by the transition function for given arguments (input symbols or the empty word). Moreover, edges of every level of such a tree are labelled with the same input symbol or with the empty word.

Remark 6.11. Formal definitions of acceptation of an input and of a language accepted by a finite automaton with ε-transitions are identical with respective definitions for deterministic and nondeterministic finite automata, c.f. Definition 6.5 and Definition 6.6.

Definition 6.15. A closure of a transitions function δ of a given finite automaton with ε-transitions $M = (Q, \Sigma, \delta, q_0, F)$ is the function:
$\hat{\delta} : Q \times \Sigma^* \to 2^Q$

1. $(\forall q \in Q)\hat{\delta}(q, \varepsilon) = \varepsilon - Cl(q)$
2. $(\forall q \in Q)(\forall a \in \Sigma)(w \in \Sigma^*)\hat{\delta}(q, wa) = \varepsilon - Cl(\delta(\hat{\delta}(q, w), a))$

where:
$\delta(P,a) = \bigcup_{p\in P} \delta(p,a)$ and $\varepsilon - Cl(P,a) = \bigcup_{p\in P} \varepsilon - Cl(p,a)$ is used for $P \subset Q$

The closure of a transition function of a finite automaton with ε-transitions extends a domain of a transition function form an input alphabet of an automaton to the set of all words over an input alphabet. The closure of a transition function applied to an initial configuration of a finite automaton immediately yields a result of computation $\hat{\delta}(q_0,w)$ of this automaton for the initial state q_0 and an input word $w = a_1 a_2 \ldots a_n$. However, a restriction of the closure of a transition function δ to the domain $Q \times \Sigma$ is not equal to δ. In fact

$$\hat{\delta}\big|_{Q\times\Sigma} = \varepsilon - Cl((\delta(\varepsilon - Cl(q),a))$$

For that reason a transition function of a finite automaton with ε-transitions and its closure are denoted by different symbols, δ and $\hat{\delta}$ respectively.

Remark 6.12. An input word w is accepted by a finite automaton with ε-transitions $M = (Q,\Sigma,\delta,q_0,F)$ if and only if $\hat{\delta}(q_0,w) \cap F \neq \varnothing$.

A question may be asked whether finite automata with ε-transitions are equivalent to deterministic ones. The answer is positive. Below we present a proof that finite automata with ε-transitions are equivalent to nondeterministic finite automata. Equivalence with deterministic finite automata comes in the form of a direct conclusion of Proposition 6.2.

It is clear that a nondeterministic finite automaton is such a finite automaton with ε-transitions, which does not use ε-transitions. Formally, a nondeterministic finite automaton can be turned to a finite automaton with ε-transitions by extending a domain of its transition function to $Q \times (\Sigma \cup \{\varepsilon\})$ and setting a set of the values of ε-transitions to the empty set.

On the other hand, we will prove that, for a given finite automaton with ε-transitions, there exists an equivalent nondeterministic one, i.e. accepting the same language. An idea of construction of such a nondeterministic automaton is based on analysis of a description of a transition given in point 2 of Remark 6.9. It could be concluded that a transition, as described there, corresponds to a transition of a nondeterministic finite automata. On the other hand, a transition described there is what closure of a transition function for an input symbol yields. Details are given as the following:

Proposition 6.2. *Finite automata with ε-transitions are equivalent to nondeterministic ones.*

Proof. First, a following finite automaton with ε-transitions:

$$M = (Q,\Sigma,\delta,q_0,F)$$

is equivalent to a nondeterministic one :

$$M = (Q,\Sigma,\delta',q_0,F)$$

such that $\delta' : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$, $\delta'(q,a) = \delta(q,a)$, $\delta'(q,\varepsilon) = \varnothing$ for $q \in Q$ and $a \in \Sigma$

Second, let us assume that now a finite automaton with ε-transitions:

$$M = (Q, \Sigma, \delta, q_0, F)$$

An equivalent nondeterministic automaton is denoted as follows:

$$M' = (Q, \Sigma, \delta', q_0, F')$$

where:

- $\delta' \equiv \hat{\delta}\,|_{Q\times\Sigma}$ - the transition function of M' is equal to the closure of the transition function of M,
- $F' = \begin{cases} F \cup \{q_0\} & if \;\; \varepsilon - Cl(q_0) \cap F \neq \varnothing \\ F & otherwise \end{cases}$.

A formal proof of equivalence of M and M' is based on mathematical induction with regard to the length of input word of both automata.

Let us prove that the closure of transition function of both automata is equal for any nonempty word $w \in \Sigma^+$

$$\delta'(q_0, w) = \hat{\delta}(q_0, w) \quad (^*)$$

Note that δ' denotes transition function of M and its closure. However, the transition function of M and its closure cannot be denoted by the same symbol.

1. for words of length 1 the equivalence $(^*)$ is derived directly from definition of both functions,
2. let us check if the equivalence $(^*)$ holds for a word $a_1 a_1 \ldots a_n a = wa$:

$$\delta'(q_0, wa) \underbrace{=}_{\substack{closure\ of \\ transition \\ function\ \delta'}} \bigcup_{p\in\delta'(q_0,w)} \delta'(p,a) \underbrace{=}_{\substack{inductive \\ assumption}} \bigcup_{p\in\hat{\delta}(q_0,w)} \delta'(p,a) \underbrace{=}_{\substack{definition \\ of\ \delta'}}$$

$$= \bigcup_{p\in\hat{\delta}(q_0,w)} \hat{\delta}(p,a) \underbrace{=}_{\substack{closure\ of \\ transition \\ function\ \hat{\delta}}} \hat{\delta}(q_0, wa)$$

3. utilizing mathematical induction based on 1. and 2. we conclude that the equivalence $(^*)$ holds.

We have just confirmed that both automata M and M' compute the same set of states for given input word $w \in \Sigma^*$. In fact, this is only a step to finalize the proof, i.e. to show that a given input word is either accepted by both automata M and M', or is rejected by them. Let us consider the following cases:

- transition functions yield the following values for the empty word: $\delta(q_0, \varepsilon) = \varepsilon - Cl(q_0)$ and $\delta'(q_0, \varepsilon) = \{q_0\}$. Thus, ε is accepted by M if and only if

$\delta(q_0,\varepsilon)\cap F\neq\varnothing$. On the other hand, $q_0\in F'$ if and only if $\varepsilon-Cl(q_0)\cap F\neq\varnothing$. Since ε is accepted by M' if and only if $q_0\in F'$, then the empty word ε is either simultaneously accepted by both automata, or is simultaneously rejected by them,

- if $\varepsilon-Cl(q_0)\cap F\neq\varnothing$ or $q_0\in F$, then we get that $F'=F$. Simultaneous acceptation is derived form the equality $(^*)$ and definitions of acceptation of an input word by M and M',
- if $\varepsilon-Cl(q_0)\cap F\neq\varnothing$ and $q_0\notin F$, then for any nonempty word $w\in\Sigma^+$ we get two subcases:
  - if $q_0\notin\delta'(q_0,w)$, then either both $\delta'(q_0,w)$ and $\hat{\delta}(q_0,w)$ include an accepting state, or none does. Again, a simultaneous acceptation is derived from the equality $(^*)$ and the definitions of acceptance of an input word by M and M',
  - the assumption $q_0\in\delta'(q_0,w)$ of this subcase implies that the word w is accepted by M'. $\delta'(q_0,w)=\hat{\delta}(q_0,w)$ due to $(*)$. We have $\varepsilon-Cl(\hat{\delta}(q_0,w))\cap F\neq\varnothing$, because $q_0\in\hat{\delta}(q_0,w)$. Let us assume that $w=ua$ for $u\in\Sigma^*$ and $a\in\Sigma$. Then, from Definition 6.15 and idempotency of $\varepsilon-Cl$ we get:

$$\varepsilon-Cl(\hat{\delta}(q_0,w))=\varepsilon-Cl(\varepsilon-Cl(\delta(\hat{\delta}(q_0,u),a)))=$$

$$\varepsilon-Cl(\delta(\hat{\delta}(q_0,u),a))=\hat{\delta}(q_{0,w})$$

    Finally, we derive that the word w is accepted by M, because $\delta'(q_0,w)=\hat{\delta}(q_0,w)\cap F\neq\varnothing$.

- no other case can be found.

We have proved that a word $w\in\Sigma^*$ is either simultaneously accepted by both automata, or is simultaneously rejected by them. This completes the proof.

6.4 Finite automata as Turing machines

In this section evidence is given that finite automata are restricted Turing machines. Furthermore, since finite automata always terminate their computation, they are restricted Turing machines with the stop property. A discussion will be focused on deterministic finite automata. However, since both classes of finite automata: nondeterministic one and with ε-transitions one are equivalent to the class of deterministic ones, the conclusions of this discussion concern all three classes.

A deterministic finite automaton can be simulated by a Turing machine, which shifts the head right and terminates computation as soon as an input word has been read. The machine accepts its input if and only if the automaton accepts it. The details of construction of a Turing machine equivalent to a deterministic finite automaton are given below.

Proposition 6.3. *There exists a Turing machine with the stop property equivalent to a given deterministic finite automaton.*

Proof. Let a deterministic finite automaton is given

$$M = (Q, \Sigma, \delta, q_0, F)$$

A Turing machine with two-way infinite tape and with halting states equivalent to this deterministic finite automaton is:

$$M_T = (Q_T, \Sigma, \Gamma_T, \delta_T, q_0, B, \{q_A\}, \{q_R\})$$

where:

- $Q_T = Q \cup \{q_A, q_R\}$ - two halting states q_A and q_R are added to the set of states Q, $q_A, q_R \notin Q$,
- $\Gamma_T = \Sigma \cup \{B\}$ - the blank symbol B and the input alphabet create the tape alphabet, $B \notin \Sigma$,
- the transition function δ_T is described with the conditions:
  $\delta_T(q,a) = (\delta(q,a), B, R)$ for $q \in Q$, $a \in \Sigma$,
  $\delta_T(q,B) = \begin{cases} (q_A, B, R) & for \quad q \in F \\ (q_B, B, R) & for \quad q \in Q - F \end{cases}$

The input configuration of Turing machine M_T is:

- an input word of the deterministic finite automaton M is stored on the tape of M_T,
- the head of the tape is placed over the first input symbol,
- the control unit of M_T is in the initial state q_0.

Of course, the Turing machine M_T terminates computation as soon as it reaches a halting state. It accepts its input if and only if the deterministic finite automaton M accepts this input.

Finite automata are special cases of pushdown automata:

Proposition 6.4. *There exists a pushdown automaton equivalent to a given deterministic finite automaton.*

Proof. Let a deterministic finite automaton is given:

$$M = (Q, \Sigma, \delta, q_0, F)$$

A deterministic pushdown automaton equivalent to this deterministic finite automaton is:

$$M_S = (Q, \Sigma, \Gamma, \delta_S, q_0, F)$$

where:

- $\Gamma = \{\triangleright\}$ - there is only one stack symbol, the initial stack symbol,
- the transition function δ_S is described with the condition:
  $\delta_S(q, a, \triangleright) = \{(\delta(q,a), \triangleright)\}$.

Chapter 7
Grammars versus automata

In Chapter 1 regular expressions and regular languages were discussed. Regular grammars were also introduced. In Chapter 6 finite automata were studied. In this Chapter it is proved that regular grammars generate and finite automata accept the class of regular languages, i.e. the class of languages generated by regular expressions. In other words, it is evident that concepts of regular expressions, regular grammars and finite automata are equivalent.

7.1 Regular expressions, regular grammars and finite automata

7.1.1 Regular expressions versus finite automata

Below we prove that regular expressions are equivalent to finite automata. First, we construct automata equivalent to regular expressions. A proof is based on the inductive definition of regular expressions, c.f. Definition 1.1 and applies mathematical induction. A constructed finite automaton equivalent to a given regular expression is an automaton with -transition. Then, a regular expression equivalent to a deterministic finite automaton is constructed. Furthermore, equivalence of finite automata and regular expression is draw on the above two constructs and on equivalence of all three classes of finite automata as shown in Figure 7.1.

Proposition 7.1. *Languages generated by regular expressions are accepted by finite automata.*

Proof. We use mathematical induction to prove this proposition. A formal proof of equivalence is done with regard to length of a regular expression, i.e. number of symbols in it.

1. there are 3 families of regular expressions of length 1:

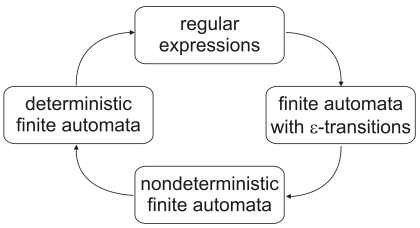

Fig. 7.1 Equivalence of finite automata and regular expressions, a dependency diagram

a. the empty regular expression Θ generates the empty language $\emptyset$. An equivalent finite automaton is shown in Figure 7.2 a
b. the empty regular expression ε generates the language with the empty word $\{\varepsilon\}$. An equivalent finite automaton is shown in Figure 7.2 b
c. a family of regular expressions $\boldsymbol{a}$, for each symbol of an input alphabet $\boldsymbol{a} \in \Sigma$, generate languages with a one letter word $\{\boldsymbol{a}\}$. An equivalent finite automaton is shown in Figure 7.2 c

Note that automata shown in Figure 7.2 are nondeterministic finite automata.

2. assume that two regular expressions $\boldsymbol{s}$ and $\boldsymbol{t}$ are given and that these expressions generate languages S and T, respectively. Assume that both expressions are given at the same alphabet Σ (otherwise, take union of alphabets of both expressions as a common alphabet). Based on inductive assumption, take finite automata M_S and M_T shown in Figure 7.3, which are equivalent to regular expressions $\boldsymbol{s}$ and $\boldsymbol{t}$, respectively. Now, we consider sum $\boldsymbol{s+t}$, concatenation $\boldsymbol{s}\circ\boldsymbol{t}$ and Kleene closure s^* of regular expressions. Languages generated by these expressions are $S \cup T$, $S \circ T$ and S^*. Finite automata (with ε-transitions), which accept union and concatenation of languages S and T, and a finite automaton accepting Klenee closure of the language S are constructed as follows:

   a. a transition diagram of a finite automaton (with ε-transitions) equivalent to sum $\boldsymbol{s+t}$ of regular expressions $\boldsymbol{s}$ and $\boldsymbol{t}$ is displayed in Figure 7.4,
   b. a transition diagram of a finite automaton (with ε-transitions) equivalent to concatenation $\boldsymbol{s+t}$ of regular expressions $\boldsymbol{s}$ and $\boldsymbol{t}$ is displayed in Figure 7.5,
   c. a transition diagram of a finite automaton (with ε-transitions) equivalent to Kleene closure $\boldsymbol{s}*$ of regular a expression $\boldsymbol{s}$ is displayed in Figure 7.6,

3. employing mathematical induction to 1. and 2. we conclude that there exists a finite automaton equivalent to any regular expression.

Remark 7.1. Let $M_S = (Q_S, \Sigma, \delta_S, q_0^S, F_S)$ and $M_T = (Q_T, \Sigma, \delta_T, q_0^T, F_T)$. The finite automaton shown in Figure 7.4 is $M = (Q, \Sigma, \delta, q_0, F)$, where:

- $Q = Q_S \cup Q_T \cup \{q_0, q_A\}$, assuming that sets of states are pair wise disjoint, i.e. $Q_S \cap Q_T = \emptyset$, $Q_S \cap \{q_0, q_A\} = \emptyset$ and $Q_T \cap \{q_0, q_A\} = \emptyset$,
- $F = \{q_A\}$,

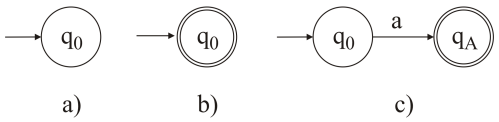

Fig. 7.2 Finite automata equivalent to basic regular expressions

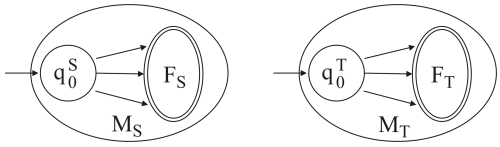

Fig. 7.3 Finite automata M_S and M_T equivalent to given regular expressions ***s*** and ***t***

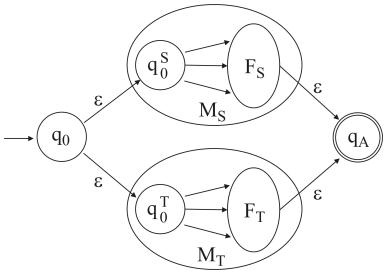

Fig. 7.4 A finite automaton equivalent to sum of regular expressions ***s*** and ***t***

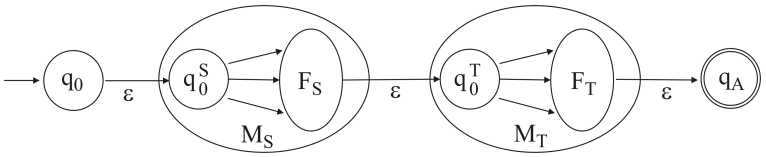

Fig. 7.5 A finite automaton equivalent to concatenation of regular expressions ***s*** and ***t***

- the transitions function δ is based on transition functions of both automata Q_S and Q_T:
  - $\delta(q,a) = \delta_S(q,a)$ for $q \in Q_S$ and $a \in \Sigma$,
  - $\delta(q,a) = \delta_T(q,a)$ for $q \in Q_T$ and $a \in \Sigma$,
  - $\delta(q,\varepsilon) = \delta_S(q,\varepsilon)$ for $q \in Q_S \setminus F_S$,
  - $\delta(q,\varepsilon) = \delta_T(q,\varepsilon)$ for $q \in Q_T \setminus F_T$,
  - $\delta(q,\varepsilon) = \delta_S(q,\varepsilon) \cup q_A$ for $q \in F_S$,
  - $\delta(q,\varepsilon) = \delta_T(q,\varepsilon) \cup q_A$ for $q \in F_T$,
  - $\delta(q,\varepsilon) = \{q_0^S, q_0^T\}$,
  - $\delta(q,\varepsilon) = \emptyset$ for such $(q,X) \in Q \times (\Sigma \cup \{\varepsilon\})$, which are not considered above.

Proposition 7.2. *Languages accepted by finite automata are generated by regular expression.*

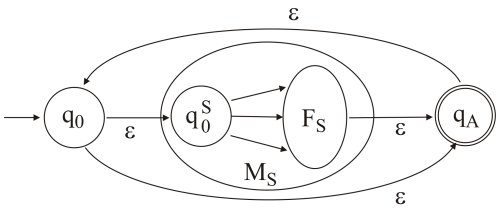

Fig. 7.6 A finite automaton equivalent to Kleene closure of a regular expression *s*

Proof. Let assume that a deterministic finite automaton

$$M = (\{q_1, q_2, \ldots, q_n\}, \Sigma, \delta, q_1, F)$$

accepts the language $L = L(M)$. A method of construction of a regular expression equivalent to the automaton M relays on construction of families of languages, which are obviously regular and which are easy to get regular expressions generating them.

The families $R^k_{i,j}$ of languages is constructed, where $R^k_{i,j}$, for given natural numbers $i, j \geq 1$ and $k \geq 0$, denotes a set of such words $w \in \Sigma^*$, that a computation for a word w, starting in the state q_i, ends in the state $q_j = \delta(q_i, w)$ and does not visit states with indexes greater than k. note that indexes i and j may be greater than k. Let us recall that a computation of a deterministic finite automaton is a sequence of alternating states and input symbols. In our case, the computations for the word w begins with the state q_i and ends with the state q_i.

Formal description of the family $R^k_{i,j}$ of languages is as follows:

$$R^0_{i,j} = \begin{cases} \{a \in \Sigma : \delta(q_i, a) = q_j\} & \text{for } i \neq j \\ \{a \in \Sigma : \delta(q_i, a) = q_j\} \cup \{\varepsilon\} & \text{for } i = j \end{cases}$$

$$R^k_{i,j} = R^{k-1}_{i,k} \circ \left((R^{k-1}_{k,k})^*\right) \circ R^{k-1}_{k,j} \cup R^{k-1}_{i,j} \quad \text{for } k > 0$$

Languages of the family $R^0_{i,j}$ include one-letter words, for which there is a transition from a state q_i to a state q_j. The empty word is included in languages (according to the rule that the empty word always allows for a transition to the same state). Note that any word longer than one cannot be included in a language of this family because a computation for such a word includes a middle state with an index greater than 0.

Languages of a family $R^k_{i,j}$, for a given $i, j, k > 0$, are assembled according to the following rules:

- a computation for a word w, which does not visit states with indexes greater than k, but visits the state q_k, may be decomposed to computations which do not visit q_k and
  - begins in q_i and ends in q_k, they form a language $R^{k-1}_{i,k}$,

- begins and ends in q_k, they produce multiple concatenations of a language $R_{k,k}^{k-1}$, i.e. they form a language $R_{k,k}^{k-1}$,
- begins in q_k and ends in q_j, they form a language $R_{k,j}^{k-1}$,

Concatenation of languages $R_{i,k}^{k-1}$, $R_{k,k}^{k-1}$ and $R_{k,j}^{k-1}$ form the first part of the union in the second formula,

- a computation of a word w, which does not visit states with indexes greater than k (except the beginning state q_i and the ending state q_j), may not visit the state q_k. In that case $w \in R_{i,j}^{k-1}$, what gives the second part of union in in the second formula.

The language $L(M)$ is a set of such words, for which computation begins in q_1, ends in $q \in F$ and may visit any state of M, i.e.

$$L(M) = \bigcup_{j:q_j \in F} R_{i,j}^{n}$$

Now, existence of regular expressions generating languages of families $R_{i,j}^{k}$ can be proved employing mathematical induction with regard to k:

- languages of the family $R_{i,j}^{0}$ are generated by the following regular expressions:
  - for $i \neq j$,
    - if $R_{i,j}^{0} = \{a_{i_1}, \ldots, a_{i_p}\}$ for $a_{i_1}, \ldots, a_{i_p} \in \Sigma$, then $r_{i,j}^{0} = a_{i_1} + \ldots + a_{i_p}$,
    - if $R_{i,j}^{0} = \emptyset$, then $r_{i,j}^{0} = \Theta$,
  - for $i = j$,
    - if $R_{i,j}^{0} = \{\varepsilon, a_{i_1}, \ldots, a_{i_p}\}$, $a_{i_1}, \ldots, a_{i_p} \in \Sigma$, then $r_{i,j}^{0} = \varepsilon + a_{i_1} + \ldots + a_{i_p}$,
    - if $R_{i,j}^{0} = \{\varepsilon\}$, then $r_{i,j}^{0} = \varepsilon$,
- based on inductive hypothesis assume that languages of a family $R_{i,j}^{k-1}$ are generated by regular expressions $r_{i,j}^{k-1}$. Notice that the formula for $R_{i,j}^{k-1}$ is an assembly of languages of a family $R_{i,j}^{k-1}$ using union, concatenation and Kleene closure. The assembly operators correspond to operations on regular expressions: sum, concatenation and Kleene closure. These notes lead to a following regular expression generating a $R_{i,j}^{k}$ language for a given indexes $i, j, k > 0$:

$$r_{i,j}^{k} = r_{i,k}^{k-1} \circ ((r_{k,k}^{k-1})^{*}) \circ r_{k,j}^{k-1} + r_{i,j}^{k-1}$$

In conclusion, the following regular expression is equivalent to the automaton M, assuming that $F = \{q_{i_1}, q_{i_2}, \ldots, q_{i_p}\}$:

$$r_{1,i_1} + r_{1,i_2} + \ldots + r_{1,i_p}$$

7.1.2 Regular grammars versus finite automata

Now we prove that regular grammars are equivalent to finite automata. First, we construct automata equivalent to right-linear grammars. Then, for given deterministic finite automaton an equivalent right-linear grammar is constructed. Afterward, it is shown that right-linear grammars are equivalent to left-linear grammars. As a result we come to a graph of equivalence of regular expressions, regular grammars and finite automata. The graph is exposed in Figure 7.7.

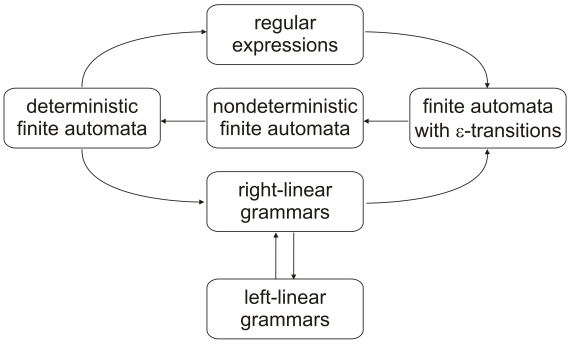

Fig. 7.7 Equivalence of finite automata, regular expressions and regular grammars, a dependency diagram

Theorem 7.1. *Languages accepted by finite automata are generated by right-linear grammars.*

Proof. Let

$$M = (Q, \Sigma, \delta, q_0, F)$$

is a deterministic finite automaton.

A right-linear grammar, which generates the language $L = L(M)$ accepted by the automaton M, is as follows

$$G(V, T, P, S)$$

where:

- $V = Q \cup \{S\}$,
- $T = \Sigma$,
- P - the set of productions includes the following productions:
  - $p \to aq$, if $\delta(p,a) = q$ for $p,q \in Q$ and $a \in \Sigma$,
  - $p \to a$, if $\delta(p,a) \in F$ for $p \in Q$ and $a \in \Sigma$,
  - $S \to q_0$,
  - $S \to \varepsilon$, if $q_0 \in F$.

Notice that a computation of the automaton M for a given word is identical with a derivation of this word in the grammar G. Based on this observation we show that a word is accepted by the automaton M if and only if it is generated by the grammar G.

First, we confirm that if a word $w \in L(M)$, then $w \in L(G)$. Let $w = a_1 a_2 \dots a_n \in \Sigma^*$. Let $q_0 a_1 q_{i_1} a_2 q_{i_2} \dots q_{i_{n-1}} a_n q_{i_n}$ is the computation of M for the word w such that $q_{i_n} \in F$, i.e. a path in a computation tree from the root to an accepting state or, in other words, $q_0 \succ^* q_{i_n}$ in a transition relation $\succ$. A corresponding derivation in G for any $w \neq \varepsilon$ is:

$$S \to q_0 \to a_1 q_{i_1} \to a_1 a_2 q_{i_2} \to \dots \to a_1 a_2 \dots a_{n-1} q_{i_{n-1}} \to a_1 a_2 \dots a_{n-1} a_n q_{i_n}$$

and then there exists a derivation of w if and only if $w \in L(M)$:

$$S \to q_0 \to a_1 q_{i_1} \to a_1 a_2 q_{i_2} \to \dots \to a_1 a_2 \dots a_{n-1} q_{i_{n-1}} \to a_1 a_2 \dots a_{n-1} a_n$$

The latter derivation applies a production $q_{i_{n-1}} \to a_n$ because $\delta(q_{i_{n-1}}, a_n) \in F$. If $\varepsilon \in L(M)$, i.e. $q_0 \in F$, then its derivation is immediate: $S \to \varepsilon$.

Now, we verify that if $w \in L(G)$, then $w \in L(M)$. Let $w = a_1 a_2 \dots a_n$, then there exists a derivation of $w\varepsilon \neq$ in G. It is as given above, i.e.

$$S \to q_0 \to a_1 q_{i_1} \to a_1 a_2 q_{i_2} \to \dots \to a_1 a_2 \dots a_{n-1} q_{i_{n-1}} \to a_1 a_2 \dots a_{n-1} a_n$$

Subsequently, a computation of M for $w = \varepsilon$ is as given above as well. The computation $S \succ q_0$, which corresponds to the derivation $S \to \varepsilon$, leads to acceptation of $w = \varepsilon$.

Theorem 7.2. *Languages generated by right-linear grammars are accepted by finite automata.*

Proof. Let

$$G = (V, T, P, S)$$

is a right-linear grammar. A finite automaton with ε-transitions accepts the language $L = L(G)$ generated by the grammar G:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where:

- $Q = \{[\alpha] : (\exists \beta \in (V \cup T)^*) A \to \beta\alpha \in P\}$, i.e. states are labelled with all possible suffixes of right hand sides of productions,
- $\Sigma = T$,
- $q_0 = [S]$,
- δ - a transition function is assembled with the rules:
  - if $A \in V$ then $\delta([A], \varepsilon) = \{[\alpha] : A \to \alpha \in P\}$,
  - if $a \in T \wedge \alpha \in (T^* \cup T^*V)$ then $\delta([a\alpha], a) = \{[\alpha]\}$

- $F = \{[\varepsilon]\}$.

Justification of correctness of the automaton construction is based on observation of a derivation in a right-linear grammar.

Any intermediate derivation word is of a form xA, where $x \in T^*$, $A \in V$. A right hand side of a production $A \to yB$ employed in a derivation replaces a nonterminal symbol A and creates a next intermediate derivation word xyB. An automaton reads terminal symbols inserted by a production and then switches to a state relevant to an inserted nonterminal symbol. This action is repeated for all productions of this form employed in a derivation. A derivation is terminated with a production $A \to z$, where $z \in T^*$ (without a nonterminal symbol in its right hand side). In this case, an automaton reads terminal symbols and then goes to an accepting state marked with the empty word ε.

A computation of such an automaton, for a given word, follows a derivation of this word. States visited along a computation correspond to unread part of an input word. This unread part is represented by beginning terminal symbols and by a one nonterminal symbol, if derivation is not terminated with a production employed. This nonterminal symbol produces a remaining part of an input word. The last production of a derivation does not include a nonterminal, what allows an automaton to read terminal symbols and to go the accepting state.

The above notes can be turn to a formal inductive proof with regard to length of derivation.

Theorem 7.3. *Right-linear grammar are equivalent to left-linear grammar.*

7.1.3 The pumping lemma

The pumping lemma was formulated in Chapter 1, but not proved there. Here it is recalled and proved.

Theorem 7.4 (the pumping lemma for regular languages).
If a language L is regular,
then there exists a constant n_L such that for any word $z \in L$ the following condition holds:

$$\left(|x| \geq n_L\right) \Rightarrow \left[\left(\bigvee_{u,v,w} z = uvw \wedge |uv| \leq n_L \wedge |v| \geq 1\right) \bigwedge_{i=0,1,2,\ldots} z_i = uv^i w \in L\right]$$

Proof. If a language L is a regular one, then there exists a deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$, which accepts L. Let denote the number of states of this automaton $|Q| = n$. If all words of the language L are shorter than n, then for the constant $n_L = n$ the implication holds because its antecedent $|z| \geq n_L$ is never true.

Let $z \in L$, $z = a_1 a_2 \ldots a_m$, where $m \geq n$. A computation of M for z is

$$q_0a_1q_{i_1}a_2q_{i_2}\ldots a_{m-1}q_{m_{m-1}}a_mq_{i_m}$$

and $q_{i_m}\in F$.

There are $m+1\geq n+1$ states in this computation. This means that at least one state is repeated, because there is only n states in M. Let us take the leftmost pair of repeating states. They, of course, appear in a beginning part of computation including no more that n letters and no more than $n+1$ states. In the following computation leftmost repeating states are underscored

$$q_0a_1q_{i_1}a_2\ldots a_j\underline{q_{i_j}}a_{j+1}q_{i_{j+1}}\ldots a_{j+p}\underline{q_{i_{j+p}}}a_{j+p+1}q_{i_{j+p+1}}\ldots a_{m-1}q_{i_{m-1}}a_mq_{i_m}$$

where $p\geq 1$.

For that reason the part of computation between these accepting states includes at least one letter. If the first underlined state together with the part between underlined (repeated) states, i.e. $\underline{q_{i_j}}a_{j+1}q_{i_{j+1}}\ldots a_{j+p}$, is removed, then the remaining sequence of states and letters is still a computation, which ends in an accepting state:

$$q_0a_1q_{i_1}a_2\ldots a_j\underline{q_{i_{j+p}}}a_{j+p+1}q_{i_{j+p+1}}\ldots a_{m-1}q_{i_{m-1}}a_mq_{i_m}$$

On the other hand, the sequence $\underline{q_{i_j}}a_{j+1}q_{i_{j+1}}\ldots a_{j+p}$ may me inserted just before the first repeated state and an obtained sequence is still a computation, which ends in an accepting state:

$$q_0a_1q_{i_1}\ldots a_j\underline{q_{i_j}}a_{j+1}\ldots a_{j+p}\underline{q_{i_j}}a_{j+1}\ldots a_{j+p}\underline{q_{i_{j+p}}}a_{j+p+1}\ldots a_{m-1}q_{i_{m-1}}a_mq_{i_m}$$

Insertion of this sequence may be repeated producing a computation, which ends in an accepting state.

As a result, the computations for following words are created:

- $z=a_1a_2\ldots a_ja_{j+p+1}\ldots a_{m-1}a_m$,
- $z=a_1a_2\ldots a_j\underline{a_{j+1}\ldots a_{j+p}}a_{j+p+1}\ldots a_{m-1}a_m$,
- $z=a_1a_2\ldots a_j\underline{a_{j+1}\ldots a_{j+p}a_{j+1}\ldots a_{j+p}}a_{j+p+1}\ldots a_{m-1}a_m$,
- $z=a_1a_2\ldots a_j\underline{a_{j+1}\ldots a_{j+p}a_{j+1}\ldots a_{j+p}a_{j+1}\ldots a_{j+p}}a_{j+p+1}\ldots a_{m-1}a_m$,
- etc.

and the above computations end in an accepting state. Notice, that a part of a word, which is repeated, has at least one letter. Moreover, both the beginning part and a repeated part are not longer than $n_L=n$. Therefore, we have a sequence of words as required in the consequent of the implication. This proves the lemma.

As a consequence of the pumping lemma, it may be concluded that computations of a finite automaton are determined by a finite set of words, which are not longer that a constant n_L. A computation for a word, which is longer than n_L, can be shortened by removing its inner part(s), as in the pumping lemma. This implies that a set of accepting states of an deterministic finite automaton can be effectively calculated by investigating a finite set of such words, which are not longer than the constant n_L.

Computations for longer words cannot bring a new accepting state. This conclusion can be formally expressed as follows.

Remark 7.2. For any word $z \in L$, $|z| \geq n_L$, there exists a word $w \in L$, $|w| < n_L$ such that the computation for the word z is $\alpha_z = \alpha_1\alpha_2\alpha_3$ and the computation for the word w is $alpha_w = \alpha_1\alpha_3$. Note that α_2 may include many different repeating parts.

7.1.4 The Myhill-Nerode Theorem

In this section the Myhill-Nerode theorem is formulated and proved. The Myhill-Nerode lemma, which was used in Chapter 1, is a direct consequence of the Myhill-Nerode theorem.

Theorem 7.5 (the Myhill-Nerode Theorem).
The following conditions are equivalent:

1. *a language L is accepted by a deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F$,*
2. *a language L is a union of some classes of a right invariant equivalence relation with finite index,*
3. *the relation R_L induced by a language L has finite index.*

Proof. The following implication between the above conditions will be shown: $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1$.

$1 \Rightarrow 2$

Assume that a deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is given. Let us define a relation $\rho_M \subset \Sigma^* \times \Sigma^*$ such that for any $x, y \in \Sigma^*$, $x \rho_M y \Leftrightarrow \delta(q_0, x) = \delta(q_0, y)$. The relation is:

- a right invariant relation because:
  $(\forall x, y, z \in \Sigma^*) \delta(q_0, x) = \delta(q_0, y) \Rightarrow \delta(q_0, xz) = \delta(q_0, yz)$,
- an equivalence relation since the equality relation is an equivalence relation. It is obvious that ρ_M is
  - reflexive, i.e. $(\forall x \in \Sigma^*)\, \delta(q_0, x) = \delta(q_0, x)$,
  - symmetric, i.e. $(\forall x, y \in \Sigma^*)\, \delta(q_0, x) = \delta(q_0, y) \;\Rightarrow\; \delta(q_0, y) = \delta(q_0, x)$,
  - transitive, i.e.
    $(\forall x, y, z \in \Sigma^*)\, \delta(q_0, x) = \delta(q_0, y)$
    $wedge \delta(q_0, y) = \delta(q_0, z) \;\Rightarrow\; \delta(q_0, x) = \delta(q_0, z)$,

It is evident that all words, for which computation ends in the same state, create equivalence class. As a conclusion, we come to the conclusion that the number of equivalence classes is equal to the number of states $|Q|$ of the automaton M. It is also evident that the language L is a union of those equivalent classes, which correspond

to accepting states.

$2 \Rightarrow 3$

For any $x, y \in \Sigma^*$, if $x\rho_M y$, then either $x, y \in L$, or $x, y \notin L$ (because L is a union of some equivalence classes of ρ_M). Moreover, for any $z \in \Sigma^*$, if $x\rho_M y$, then $xz\rho_M yz$, i.e. either $xz, yz \in L$, or $xz, yz \notin L$, (because is a right invariant relation). For that reason $(\forall x, y \in \Sigma^*) x\rho_M y \Rightarrow xR_L y$. As a conclusion, we have that every equivalence class of the relation ρ_M is included in some equivalence class of the relation R_L induced by the language L, Then we get that R_L has no more equivalence classes than ρ_M has, i.e. the number of equivalence classes of R_L is finite.

$3 \Rightarrow 1$

Assume that the relation R_L induced by the language L has finite number of equivalence classes. The following deterministic finite automaton accepts the language L:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where:

- $Q = \{q_{[w]} : w \in \Sigma^*\}$ - states correspond to equivalence classes of R_L,
- Σ - an alphabet of the language L,
- $q_0 = q_{[\varepsilon]}$ - a state corresponding to the equivalence class $[\varepsilon]$, which includes the empty word ε, is the initial state,
- $F = \{q_{[w]} : w \in L\}$ - accepting states correspond to equivalence classes, which are included into the language L,
- δ - a transition function is defined by the formula $\delta(q_{[w],a} = q_{[wa]}$, for any $q_{[w]} \in Q$ and any $a \in \Sigma$, where $[w]$ is an equivalence class of the relation R_L represented by a word $w \in \Sigma^*$.

The automaton M constructed above accepts the language L because:

- for any $w \in L$, $\delta(q_0, w) = \delta(q_{[\varepsilon],w} = q_{[w]}$ (simple inductive proof justifies this evidence), i.e. $\delta(q_0, w) \in F$,
- likewise, for any $w \notin L$, $\delta(q_0, w) \notin F$.

The proof is completed.

7.1.5 Minimization of deterministic finite automata

The Myhill-Nerode theorem permits to minimize deterministic finite automaton. First of all, note that the relation R_L induced by a regular language L is a most general equivalence relation defining a language L. Namely, an equivalence relation defines a language if and only if a language is a union of some equivalence classes of this relation. Recall, that an equivalence relation E_1 is more general than an

equivalence relation E_2 if and only if equivalence classes of E_2 are included into equivalence classes of E_1.

Secondly, a deterministic automaton constructed in the proof of the third implication is a minimal one with regard to number of states. If this would not be true, then we would be able to build a deterministic automaton M', which has less states than the automaton M constructed in the proof. However, the relation ρ_M would have less equivalence classes than the relation R_L. But this is not possible due to the second implication considered in the proof of the Myhill-Nerode theorem.

Finally, it is obvious, that an automaton constructed in the proof of the Myhill-Nerode theorem, is a minimal one with regard to the number of states.

7.2 More grammars and automata

7.2.1 Context-free grammars versus pushdown automata

In this section relations between context-free grammars and pushdown automata are discussed. It is shown that pushdown automata are equivalent to context-free grammars. Thus, a class of languages accepted by pushdown automata is the class of context-free languages.

Theorem 7.6. *Languages generated by context-free grammars are accepted by push-down automata.*

Proof. Let a context-free grammar is given and the empty word is not generated in the grammar. We construct a pushdown automaton accepting the language generated by this grammar. The automaton accepts with the empty stack.

We assume that a given context-free grammar

$$G = (V, T, P, S)$$

is in Greibach normal form. Let us recall that productions of a grammar in Greibach normal form are $A \to a\alpha$, where $A \in V$, $a \in T$ and $\alpha \in V^*$. For a given word $w \in L(G)$ a leftmost derivation in G is considered. A pushdown automaton, when computes a given word w, follows this leftmost derivation in G. A pushdown automaton equivalent to the grammar G is:

$$M = (\{q_0, q\}, T, V \cup \{\rhd\}, \delta, q, \rhd, \emptyset)$$

where the transition function is constructed as follows:

- begin with a given word $w \in T^*$ at the input of M and with the initial symbol S of the grammar G on the stack,
- accept if the end-of-input symbol $\lhd$ is at the input and the initial stack symbol $\rhd$ is on the stack,

- if $a \in T$ is an input symbol, $A \in V$ is a top symbol of the stack and there is a production $A \to a\alpha$ in the grammar G, $\alpha \in V^*$, then read the input symbol and replace the top symbol of the stack with α (remove A form the stack, push on the stack symbols of α in reverse order,
- reject in all other cases.

These rules could be re-written as follows:

- $\delta(q_0, \varepsilon, \rhd) = \{(q, S\rhd)\}$,
- $\delta(q_0, a, A) = \{(q, A) : A \to a\alpha \in P\}$,
- $\delta(q, \lhd, \rhd) = \{(q, \varepsilon)\}$.

Modification of the construction for the case when ε is included in the language is fairly easy. The automaton should be able to pop the top symbol of the stack up in its first transition, i.e., the first rule of the presented above should be replaced with:

- $\delta(q_0, \varepsilon, \rhd) = \{(q, S\rhd), (q, \varepsilon)\}$,

A formal proof is based on mathematical induction with regard to length of derivation.

Notice that an automaton constructed in Theorem 7.6 is, in general, nondeterministic one. Nondeterminism is raised by ambiguity of a grammar. If a grammar in Greibach normal form is simple, i.e. it satisfies the Greibach uniqueness condition, then an automaton is a deterministic one.

Theorem 7.7. *Languages accepted by push-down automata are generated by context-free grammars.*

Proof. Assume that an automaton is given $M = (Q, \Sigma, \Gamma, \delta, q_0, \rhd, F$. A grammar that is equivalent to M should keep track of transitions made my M. Intuitively, a computation should be followed by a left-most derivation in a grammar in Greibach normal form. Every transition of an automaton should be accompanied by a set of productions. Productions are containers to store states and symbols employed in transitions. Left hand side of such a production is a top symbol of the stack accompanied with states before the transition is done and after it is done. Right hand side of it is a sequence of symbols, which are put on the stack. These symbols are accompanied with states, that are employed in a transition with this stack symbol and these states. This sequence is preceded by a terminal symbol, which is an input symbol consumed in this transition.

Now, let us give a detailed description of the grammar $G = (V, T, P, S)$

- $V = Q \times \Gamma \times Q \cup \{S\}$,
- $T = \Sigma$,
- productions are:
  - for all $S \to (q_0, \rhd, q)$ for all$q \in Q$, these productions initiate simulation of a computation,

- if $\delta(q,\alpha,X)$ includes $(r,A_1,A_2,\ldots,A_m)$ for $\alpha \in \Sigma \cup \{\varepsilon\}$, $q,r \in Q$ and $X,A_1,A_2,\ldots,A_m \in \Gamma$, then the following productions should be added for all $r_1,r_2,\ldots,r_m \in Q$: $(q,X,r_n) \to a(r,A_1,r_1)(r_1,A_1,r_2)\ldots(r_{n-1},A_1,r_n)$, i.e. a transition $(r,A_1,A_2,\ldots,A_m)) \in \delta(q,\alpha,X)$ is developed in such a way, that (1) a state r after the first switch from a state Q is preserved in all productions, (2) a final state r_n after employing the series of transitions for stack symbols $A_1,A_2,\ldots,A_m \in \Gamma$ is preserved as well and (3) a track of states along the series of transitions is kept,
- if $\delta(q,\alpha,X)$ includes (r,ε) for $q,r \in Q$ and $\alpha \in \Sigma \cup \{\varepsilon\}$, i.e. a top symbols of the stack is pop out, then a corresponding nonterminal symbols from an intermediate word of a derivation: $(q,X,r) \to \alpha$.

Chapter 8
The hierarchy

8.1 More operations on languages

8.1.1 Substitutions, homomorphisms

Definition 8.1. Let Σ and Δ are alphabets. A mapping f of letters of an alphabet Σ into languages over an alphabet Δ:

$$f : \Sigma \to 2^{\Delta^*}$$

is called substitution. A substitution on an alphabet Σ can be extended

- to words over an alphabet Σ:

$$\begin{aligned} f &: \Sigma^* \to 2^{\Delta^*} \\ f(\varepsilon) &= \varepsilon \\ f(wa) &= f(w) \circ f(a) \ \ (\forall w \in \Sigma^*)(\forall a \in \Sigma) \end{aligned}$$

- and to languages over an alphabet Σ:

$$\begin{aligned} f &: 2^{\Sigma^*} \to 2^{\Delta^*} \\ f(L) &= \bigcup_{w \in L} f(w) \ \ (\forall \subset L) \end{aligned}$$

This general definition of a substitution may be restricted to a class of languages assuming that values of a substitution are languages of this class as well as arguments of a substitution are languages of this class as well. In this section discussion on substitutions is restricted to regular languages. Explicitly, we assume that substitutions are mappings: $f : \Sigma \to RgL(\Delta)$, $f : \Sigma^* \to RgL(\Delta)$ and $f : RgL(\Sigma) \to RgL(\Delta)$, where $RgL(\Sigma)$ and $RgL(\Delta)$ are the classes of regular languages over alphabets Σ and Δ, respectively.

The definition of a substitution could be reformulated to regular expressions.

Definition 8.2. Let Σ and Δ are alphabets. A mapping F of letters of an alphabet Σ into regular expressions $REx(\Delta)$ over an alphabet Δ:

$$f : \{a : a \in \Sigma\} \to REx(\Delta)$$

A substitution on an alphabet Σ can be extended to regular expressions $REx(\Sigma)$ over an alphabet Σ:

$$f : REx(\Sigma) \to REx(\Delta)$$

such that

$$\begin{aligned} f(\emptyset) &= \emptyset \\ f(\varepsilon) &= \varepsilon \\ f(a) &= \text{according to definition of f} \quad (\forall a \in \Sigma) \end{aligned}$$

and

$$\begin{aligned} f(s+t) &= (f(s)+f(t)) \\ f(s \circ t) &= (f(s) \circ f(t)) \\ f(s^*) &= \left((f(s))^*\right) \end{aligned}$$

where $\emptyset$, ε and a for $a \in \Sigma$ are basic regular expressions, s and t are regular expressions (used in inductive step of the definition).

Remark 8.1. In the class of regular languages, substitutions can be considered alternatively with regard to languages or with regard to regular expressions generating these languages.

Definition 8.3. Let Σ and Δ are alphabets. A substitution $h : \Sigma \to 2^{\Delta^*}$, such that $|h(a)| = 1$ for all $a \in \Sigma$, is called a homomorphism. In other words, a homomorphism is such a substitution, which yields one word languages. An extension of a homomorphism to words and languages is as a relevant extension of a substitution.

Remark 8.2. A homomorphism h is identified with a mapping $h : \Sigma \to \Delta^*$, which yields a words over an alphabet Δ for a letter of an alphabet Σ rather than a languages including only this word.

Definition 8.4. Let $h : \Sigma \to 2^{\Delta^*}$ is a homomorphism. An inverse homomorphic image of a word $z \in \Delta^*$ is a set of words (language):

$$h^{-1}(w) = \{x \in \Sigma : h(x) = w\}$$

An inverse homomorphic image of a language $L \subset \Delta^*$ is a set of words:

$$h^{-1}(L) = \bigcup_{w \in L} h^{-1}(w) = \{x \in \Sigma^* : h(x) \in L\}$$

8.1.2 Quotients

Quotients of languages are actually applied to words. Quotient of words is essentially the opposite of concatenation. A quotient of words is a kind of reduction of one word, a dividend, by another one, a divisor. Two types of quotients are defined: the right quotient and the left quotient.

Definition 8.5. Let L_1 and L_2 are languages over an alphabet Σ.

The right quotient of a language L_1 with a language L_2 is the language L_1/L_2 consisting of such words over the alphabet Σ words, which concatenated with words of the divisor give words of the dividend:

$$L_1/L_2 = \{x \in \Sigma^* : (\exists y \in L_2)xy \in L_1\}$$

The left quotient of a language L_1 with a language L_2 is the language $L_1 \backslash L_2$ consisting of such words over the alphabet Σ words, which concatenated to words of the divisor give words of the dividend:

$$L_1 \backslash L_2 = \{x \in \Sigma^* : (\exists x \in L_2)xy \in L_1\}$$

Remark 8.3. The definition of quotients of languages could be reformulated to regular expressions instead of languages. Such a formulation corresponds, of course, to quotients of languages generated by regular expressions, i.e. to quotients of regular languages.

8.1.3 Automata building with quotients

A language $L = L(M)$ accepted by a deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is a set of words $L = \{w \in \Sigma^* : \delta(q_0, w) \in F\}$. Let consider languages:

- $L_a = \{w \in Sigma^* : \delta(q_a, w) \in F\}$, where $\delta(q_0, a) = q_a$ for $a \in \Sigma$. These languages are accepted by deterministic automata $M = (Q, \Sigma, \delta, q_a, F)$. Note that L_a is derived form L by removing the first letter from words of L, i.e. $L_a = \{u \in Sigma^* : (\exists w \in L)\, au = w\}$. The last formula defines the left quotient of the language L with the language $\{a\}$ (this language includes one word of unit length). i.e. $L_a = L \backslash \{a\}$,
- $L_{ab} = \{w \in Sigma^* : \delta(q_{ab}, w) \in F\}$, where $q_{ab} = \delta(q_0, ab)$ for $a, b \in \Sigma$. L_{ab} is derived form L by removing two leading first letters from words of L or - in other words - L_b is derived from L_a by removing the first letter from words of it: $L_{ab} = L \backslash \{ab\} = L_a \backslash \{b\}$,
- $L_{abc} = \{w \in Sigma^* : \delta(q_{abc}, w) \in F\}$, $\delta(q_0, a) = q_{ab}$ for $a, b, c \in \Sigma^*$,
- $ldots$.

How many languages do we get in the above process of quotients? As many as words in the language L, at a glance. But, since languages are tied to states of a finite

automaton, we get no more languages that the number of states. On the other hand, languages are computed as quotients by words over the alphabet of this language and may be tied to any deterministic finite automaton including an automaton with minimal number of states. Hence, the number of languages does not depend on a particular automaton. From the Myhill-Nerode theorem we conclude that these languages are tied with equivalent classes of the relation R_L induced by a regular language L rather then with a particular deterministic finite automaton accepting L. However, these languages are not equivalence classes of R_L. Moreover, they are not equivalence classes of any equivalence relation.

Now, consider which of the above languages correspond to equivalence classes of R_L included in the language L. Take a particular language L_u. Note that there are many $u \in \Sigma^*$ defining the same language. In fact, a set of words defining a particular language can be written as $E_w = \{u \in \Sigma^* : L_u = L_w\}$. If an automaton M is a minimal one, then $E_w = \{u \in \Sigma^* : \delta(q_0,u) = \delta(q_0,w)\}$ is an equivalence class of R_L, c.f. the Myhill-Nerode theorem. If v is a shortest word in E_w, then the state $\delta(q_0,v) = \delta(q_0,w)$ is accepting one if and only if $v \in L$, what is equivalent to $\varepsilon \in L_w$.

As a conclusion of this discussion, we get the following proposition:

Proposition 8.1. *A regular language L over an alphabet Σ^* is accepted by a deterministic finite automaton*

$$M = (Q, \Sigma, \delta, q_0, F)$$

where:

- $Q = \{q_{L_w} : L_w = L\backslash\{w\} \wedge w \in \Sigma^*\}$, *i.e. states are labelled by quotients of languages,*
- $q_0 = q_L$,
- $F = \{q_{q_{L_w}:\varepsilon\in L_w}$,
- $\delta(q_{L_w}, a) = q_{L_{wa}}$ *for* $a \in \Sigma$, $w \in \Sigma\Sigma^*$.

Moreover, this automaton is minimal one (with regard to number of states) accepting L assuming that all equal languages are identified.

8.2 The hierarchy of languages

Proposition 8.2. *The class RgL of regular languages in included, but not equal to the class CFL of context-free languages.*

Proof. According to Theorem 7.1 and Theorem 7.3 regular languages are generated by right-linear grammars. On the other hand, right-linear grammars are context-free ones. Thus, we have inclusion. Moreover, the language $L = \left\{w \in \{a,b\}^* : \#_a w = \#_b w > 0\right\}$ is not a regular one, but it is a context-free one.

Proposition 8.3. *The class CFL of context-free languages in included, but not equal to the class CSL of context-sensitive languages.*

Proof. Again, context-free languages are generated by context-free grammars. On the other hand, context-free grammars are also context-sensitive ones, which generate context-sensitive languages. Moreover, the language $L = big\{w \in \{a,b,c\}^* : \#_a w = \#_b w = \#_c w > 0\}$ is not a context-free one, but it is a context-sensitive one. Thus, we have inclusion, but not equality.

Lemma 8.1. *There is a Turing machine with the stop property (an algorithm) to check if a given word $z = a_1 a_2 \ldots a_n$ is generated by a given context-sensitive grammar $G = (V,T,P,S)$.*

Proof. The Turing machine realizes a shortest paths algorithm. Let us build a graph with nodes labelled by all words $w \in (V \cup T)^*$, which are not longer than n. Note that the initial symbol of the grammar S and the word z are among labels of nodes. Two nodes labelled with words u and v are connected with a directed edge if and only if there is a direct derivation of v from u in G. In this way, $w \in L(G)$ if and only if there is a path in this graph from the node labelled S to the node labelled z. A Turing machine with the stop property could be built, which checks existence of such a path and find the path. Such a machine implements an algorithm for path searching in a graph.

Lemma 8.2. *The set of context-sensitive grammars is countable, i.e. context-sensitive grammars can be enumerated with natural numbers.*

Proof. In fact, we can encode context-sensitive grammars as natural numbers. Assume that $G = (V,T,P,S)$ is a context sensitive grammar. Then the following is done:

- terminal and nonterminal symbols are replaced with binary numbers represented by a blocks of digits of fixed lengths. The number of binary digits necessary for encoding terminal and nonterminal symbols and an additional symbol is equal to $p = \lfloor \log_2(|V| + |T| + !) \rfloor + 1$. Assume that the number $2^p - 1$ enumerates a special symbol, a separator. It represented as the block $111\ldots11$ of p binary digits 1,
- nonterminal symbols are enumerated by successive natural numbers starting with 0 (represented as the block $000\ldots00$ of p binary digits 0). Assume that the beginning symbol S of a grammar is enumerated with 0. Of course, every number is represented by a string of p binary digits, some or all of them with nonsignificant zeros,
- enumeration of terminal symbols is continued with successive natural numbers following enumeration of nonterminal symbols,
- productions are represented as sequences of p-digits blocks enumerating symbols. The special symbol (i.e. the block of p ones) separates both hand sides of productions,
- a grammar is encoded as the following sequence of blocks of p binary digits:
  - encoding begins with two separators (two blocks of 1s),
  - nonterminal symbols ($|V|$ blocks of binary digits, the first one is the block of 0s),

- the separator,
- terminal symbols ($|T|$ blocks of binary digits),
- the separator and a production, these two elements are repeated for every production,
- encoding ends with two separators.

As a result, we get a binary number that encodes the given grammar G.

Note that at least one context-sensitive grammar can be encoded as a given natural number and not every number encodes a grammar, i.e. such numbers, for which binary representation is not valid code of any grammar. However, we can assume that numbers, which are not valid codes of context-sensitive grammars, encode a grammar generating the empty language. Likewise, not every natural number represents a word over the set of terminal symbols T, for instance, binary words shorter than p. But we can treat such natural numbers as not generated by the grammar.

This encoding is ambiguous, i.e. a given grammars can have many codes. For instance, an order of symbols or productions affects the result of encoding. Anyway, all context-sensitive grammars are encoded as natural numbers and no number is a code of two grammars. Therefore, grammars can be ordered according to the smallest codes, what gives a method of enumeration of grammars and accessing the grammar encoded as a given number. Simply, take binary representation of successive natural numbers and then check, if it is a correctly encoded grammar. If a grammar of a given code is searched, continue this process until this code is found. Of course, any grammar can be identified in this way.

Proposition 8.4. *The class CSL of context-sensitive languages in included, but not equal to the class RkL of recursive languages.*

Proof. Context-sensitive languages are accepted by linear bounded automata. Linear bounded automata are restricted Turing machines with the stop property. Recursive languages are accepted by Turing machines with the stop property. Thus, we have inclusion of the class CSL in the class RkL.

Now we construct a language that is in RkL class, but not in CSL class. Let us build a relation $r \subset N \times N$. A pair (k,l) of natural numbers belongs to this relation if and only if the context-sensitive grammar encoded as l generates the binary word at kth place in the canonical order, i.e. $r_{k,l} = 1$ if the lth grammar generates the kth word, $r_{k,l} = 0$ otherwise. Consider the language of words, which are not generated by the corresponding grammar, i.e. with 0 at the main diagonal in Table 8.1. This is so called diagonal language $L_d = \{w \in \{0,1\}^* : w = w_i \wedge r_{i,i} = 0big\}$ in the class CSL. We will come to contrary, if we assume that L_d is context-sensitive. If it is context-sensitive, then - due to Lemma 8.2 - it is generated by a context-sensitive grammar encoded as some natural number, say number k and denote it G_k. Consider the word w_k in canonical order. If it is in L_d, then $r_{k,k}$ by definition of L_d. However, $r_{k,k}$ means that G_k does not generate w_k, though it should. On other hand, if w_k does not belong to L_d, then $r_{k,k}$ by definition of L_d. However, $r_{k,k}$ means that G_k generates w_k, though it should not. Thus, the diagonal language L_d in the class CSL cannot be a context-sensitive one.

The language L_d is accepted by a Turing machine with the stop property. Such a machine realizes the following algorithm:

- finds the number k of a given binary word in canonical order, i.e. $w = w_k$,
- finds the context-sensitive grammar G_k encoded as the number k,
- checks, if the grammar G_k generates the word $w = w_k$ or not. The method shown in Lemma 8.1 can be employed for checking.

This method allows answering the question, if any word is generated by a given grammar or not. This shows that the diagonal language L_d in the class CSL belongs to the RkL class. In this way we have proved that the CSL class is included, but not equal to the RkL class.

$\delta(q_0)$	0	1	2	3	...	k	...
$w_0 = \varepsilon$	$r_{0,0}$	$r_{0,1}$	$r_{0,2}$	$r_{0,3}$		$r_{0,k}$	
$w_1 = 0$	$r_{1,0}$	$r_{1,1}$	$r_{1,2}$	$r_{1,3}$		$r_{1,k}$	
$w_2 = 1$	$r_{1,0}$	$r_{1,1}$	$r_{1,2}$	$r_{1,3}$		$r_{1,k}$	
$w_3 = 00$	$r_{1,0}$	$r_{1,1}$	$r_{1,2}$	$r_{1,3}$		$r_{1,k}$	
...							
w_k	$r_{k,0}$	$r_{k,1}$	$r_{k,2}$	$r_{k,3}$		?	
...							

Table 8.1 The membership table for context-sensitive grammars.

Lemma 8.3. *The set of Turing machines is countable, i.e. Turing machines can be enumerated with natural numbers.*

Proof. The proof is similar to the proof of Lemma 8.2. We encode Turing machines as natural numbers. Assume that $M = (Q, \Sigma, \Gamma, \delta, q_0, B, \{q_A\})$ is a Turing machine with a halting accepting state. Then we encode the machine M as a binary number in the following way:

- states, symbols of the tape alphabet Γ, symbols of the input alphabet Σ and two symbols of directions of the head move are replaced with binary numbers represented by blocks of digits of fixed length. An additional symbol, a separator, is included in the set of codes. The number of binary digits necessary for such an encoding is equal to $p = \lfloor \log_2(\Sigma|| + |\Gamma| + |Q| + 3) \rfloor + 1$. Assume that the number $2^p - 1$ enumerates a special symbol, a separator. It represented as the string $111\ldots11$ of p binary digits 1,

- states are enumerated by successive natural numbers starting with 0. Assume that the initial state is enumerated with the number 0 and the accepting state - with the number 1 (of course, every number is represented by a block of p binary digits, some or all of them with nonsignificant zeros),
- enumeration of symbols of Γ is continued with successive natural numbers following enumeration of states,
- then enumeration of symbols of Σ is continued with successive natural numbers following enumeration of symbols of Γ,
- " then enumeration of directions of the head move is done with next two successive natural numbers following enumeration of states,
- every transition $\delta(q,X) = (p,Y,D)$ is represented as five p-digits blocks, namely these blocks represent arguments of the transition function (a state q and a tape symbol X) and result (a state p, a tape symbol Y and a direction D,
- a Turing machine is encoded as the following sequence of blocks of p binary digits:
  - the encoding begins with two separators (two blocks of 1s),
  - states ($|Q|$ blocks of binary digits, the first one is the block of 0s), recall that the initial state is encoded with the number 0, the accepting state is encoded - with the number 1,
  - the separator,
  - symbols of Γ, ($|\Gamma|$ blocks of binary digits),
  - the separator,
  - symbols of Σ, ($|\Sigma|$ blocks of binary digits),
  - the separator,
  - directions of the head moves (2 blocks of binary digits),
  - the separator,
  - the separator and a transition, these two elements are repeated for every transition (every entry of the transition table),
  - the encoding ends with two separators.

This encoding is ambiguous, as a similar encoding in Lemma 8.2. Natural numbers, which are not valid codes of a Turing machine, are assumed to be codes of a machine falling in infinite computation for every input. Binary words not representing any word over Σ are assumed to not be accepted by a Turing machine.

Finally, we can conclude that all Turing machines can be enumerated with natural numbers.

Proposition 8.5. *There are languages, which are not recursively enumerable.*

Proof. The set of all words Σ^* over a given alphabet Σ in infinite and countable. The class of languages $ALL = \{L : L \subset \Sigma^*\}$ is the power set of Σ^*, so then it is uncountable. On the other hand, languages of the *REL* class are accepted by Turing machines. The set of Turing machines is countable, c.f. Lemma 8.3, so the class *REL* of languages is countable. Since the class ALL is an uncountable set, then it cannot be equal to its countable subset. This proves that are languages, which are not recursively enumerable.

Now, let construct a language that is not recursively enumerable. Let $r \subset N \times N$ is a relation build in a similar way as in Proposition 8.4, i.e. a pair (k,l) of natural numbers belongs to this relation if and only if the Turing machine encoded as l accepts the binary word at kth place in the canonical order. In other words, $r_{k,l} = 1$ if the lth Turing machine accepts the kth word, $r_{k,l} = 0$ otherwise. Note, that $r_{k,l} = 0$ means that the lth Turing machine either terminates computation and rejects its input or it is doing infinite computation. Note that the diagonal language $L_d = \{w \in \{0,1\}^* : w = w_i \wedge r_{i,i} = 0\}$ in the class REL built on this relation is not recursively enumerable. Again, as in Proposition 8.4, we will come to contrary, if we assume that L_d is recursively enumerable one. If L_d is in REL class, then there exists a Turing machine, which accepts it, say the kth machine M_k. Consider $r_{k,k}$. If $r_{k,k} = 1$, then w_k is accepted by M_k, so $w_k \in L_d$, but it should not due to definition of L_d. If $r_{k,k} = 0$, then w_k is not accepted by M_k, so $w_k \notin K_d$, but it should due to definition of L_d. Thus, the diagonal language L_d cannot be a recursively enumerable one.

Lemma 8.4. *The universal language L_u is a recursively enumerable one.*

Proof. We construct a Turing machine, which accepts the universal language L_u. This is so called the universal Turing machine M_u. The machine has three tapes. It realizes the following algorithm:

1. checks if an input word is of a form $\langle M \rangle w$, where $\langle M \rangle$ is a valid code of a Turing machine,
   - looks for beginning sequence of 1s, if it is of even length 2r, rejects the input,
   - stores r 0s on the third tape, content of the third tape is used as a measure of length of the blocks of binary digits encoding the machine and as a number of current state,
   - looks for the next sequence of 2r 1s,
   - moves the beginning part of the input word bounded by both blocks of 2r 1s to the second tape, leaves w on the first tape, places the head of the first tape on the leftmost symbol of w,
   - checks if content of the second tape is a valid code $\langle M \rangle$ of a Turing machine,
2. repeats the following actions until the number stored on the third tape encodes the accepting state,
   - for the state q stored on the third tape and for the symbol X stored as a block of r binary digits with the head of the first tape placed on the leftmost digit of this block retrieve a matching transition $\delta(q,X) = (p,Y,D)$. There may be more than one matching transition, so this is nondeterministic choice,
   - replace content of the third tape with p, replace X by Y and move the head of the first tape to neighboring clock in direction described by D.

Lemma 8.5. *The universal language L_u is not a recursive one.*

Proof. Let assume that L_u is recursive, i.e. that a Turing machine M' with the stop property accepts L_u. Based on this assumption the machine M_d accepting the diagonal language in the class REL can be build, what contradicts Proposition 8.5. Therefore, the universal language cannot be recursive.

A hypothetical Turing machine M_d, assumed to accept the diagonal language in the class REL, realizes the following algorithm:

- for a given input word w, M_d retrieves the index k of w in the canonical order, i.e. it finds such i, that $w = w_i$,
- M_d retrieves binary representation $\langle M_k \rangle$ of the kth Turing machine M_k,
- M_d concatenates binary representation $\langle M_k \rangle$ of the kth Turing machine M_k with w,
- M_d simulates computation of the hypothetical machine M' for the concatenation of $\langle M_k \rangle$ and w,
- M_d terminates computation if and only if M' terminates its computation, then M_d reverses an output of M', i.e. M_d accepts if and only if M' rejects.

Note that M_d accepts if and only if the Turing machine encoded as k rejects $w = w_k$. Moreover, M_d has the stop property since the hypothetical Turing machine M' is assumed to have the stop property.

Remark 8.4. The following inclusions hold based on discussion in this section

$$RgL \subset CFL \subset CSL \subset RkL \subset REL \subset ALL$$

The following inclusions are called the Chomsky hierarchy

$$RgL \subset CFL \subset CSL \subset REL \subset ALL$$

where $\subset$ denotes inclusion, but not equality.

8.3 Closeness

In this section we will consider classes of languages examined in the previous section: regular, context-free, context-sensitive, recursive, recursively enumerable classes of languages, i.e. RgL, CFL, CSL, RkL, REL classes, and the class of all languages, the ALL class.

Let us examine this problem from two points of view. The first attempt employs grammars, which generate languages of relevant classes. This attempt allows showing that union is an inner operation in all classes of languages generated by grammars, i.e., in RgL, CFL, CSL, RkL, REL classes of languages. The second attempt is based on automata. It allows for proving union to be an inner operation in RgL, CFL, CSL, REL classes. We give proofs for both attempts, despite that in this way proofs are duplicated for some classes. We think that these proofs play utilitarian roles for discussion on closeness as well as they provide useful techniques, which could be applied in solving other problems.

Remark 8.5. Taking a subset of a given set is the very basic operation. However, it is not an inner operation in any class of languages, except the ALL class. It means

that a subset of a language of any class may not belong to this class, besides the *ALL* class.

Remark 8.6. The *ALL* class is closed with regard to any operation. Therefore, further discussion does not take this class into account.

Proposition 8.6. *Union is an inner operation in all classes of languages.*

Proposition 8.7. *Concatenation is an inner operation in all classes of languages.*

Proposition 8.8. *Kleene closure is an inner operation in all classes of languages.*

Proposition 8.9. *Substitution is an inner operation in RgL, CFL, CSL and REL classes of languages.*

Proposition 8.10. *Intersection is an inner operation in RgL, CSL, RkL and REL classes of languages.*

Proposition 8.11. *Intersection is not an inner operation in CFL class of languages.*

Proposition 8.12. *Complement is an inner operation in RgL, CSL and RkL classes of languages.*

Proposition 8.13. *Complement is not an inner operation in CFL class.*

Proposition 8.14. *Complement is not an inner operation in REL class.*

Proposition 8.15. *The inverse homomorphic image is an inner operation in the class of regular languages.*

Proposition 8.16. *Quotients (left and right) are an inner operation in the class of regular languages.*

Table 8.2 Closeness of operations on languages

is inner in	*RgL*	*CFL*	*CSL*	*RkL*	*REL*	*ALL*
union	+	+	+	+	+	+
concatenation	+	+	+	+	+	+
Kleeneclosure	+	+	+	+	+	+
substitution	+	+	+		+	+
homomorphism	+	+	+		+	+
intersection	+	−	+	+	+	+
complement	+	−	+	+	−	+
inv. hom. image	+					+
quotients	+					+

Index